CHAPTER

Electronic system-level design and high-level synthesis

5

Jianwen Zhu
University of Toronto, Toronto, Canada

Nikil Dutt
University of California, Irvine, California

ABOUT THIS CHAPTER

System designers conceptualize designs at an abstract functional level where outputs are typically described as (algorithmic or transfer) functions of the system inputs. This design abstraction level, called ***electronic system level*** (ESL), enables ease of design capture and early design space exploration of multiple design implementation alternatives. ESL designs can be refined into lower levels of abstraction through a number of steps that gradually map abstract functions into ***register-transfer level*** (RTL) components. An enabling technology for ESL design is ***high-level synthesis*** (HLS), also known as behavioral synthesis. A high-level synthesis tool bridges the gap between an algorithmic description of a design and its structural implementation at the register transfer level and is the next natural step in design automation, succeeding logic synthesis. The need for high-level synthesis becomes more pressing with the proliferation of billion-transistor designs. High-level synthesis has been researched actively since the 1980s, and has yielded several promising results. However, it also faces a number of challenges that has prevented its wide adoption in practice.

In this chapter, we first introduce the notion of ESL design and an ESL design method. Next, we describe high-level synthesis in the context of an ESL design method. We then describe the generic structure of the high-level synthesis process and the basic tasks accomplished by high-level synthesis. This is followed by a detailed description of the key high-level synthesis algorithms and exercises designed to reinforce understanding. The reader will have been exposed to the basic principles of high-level synthesis and its applicability in an ESL design flow by the end of the chapter.

5.1 INTRODUCTION

A key goal of ***electronic design automation*** (EDA) is to shrink the rapidly growing "designer productivity gap" that exists between how many transistors we can manufacture per chip, and how many person-years we need to complete a design with that many transistors. Collectively, EDA provides to chip designers a *design method*, which can be considered as a set of complementary *design tools* built on a *design abstraction* (*i.e.*, a mechanism to conceptualize the chip design), as well as a set of processes and guidelines that indicate the flow of design, ordering of tool application, strategies for incorporating late engineering changes, etc. The software design tools include *design entry* tools, which capture design specification; *design synthesis* tools, which target different parts of the design specification and bring them down to low level implementation; and *design verification* tools, which either simulate/verify the specification or compare a specification against its implementation.

As discussed in Chapter 1, the EDA design method has traditionally progressed by raising the abstraction at which designs are conceptualized and specified. As a step in this progression, the basic components used by designers grow in complexity, which results in fewer, but more complex, components; therefore, designer productivity improves because designers need to manipulate fewer components and can reason about the design at abstractions that are closer to how systems are specified and conceptualized. Historically, the basic design component has evolved from polygons, to transistors, to gates, and then to register transfer level blocks. To cope with the challenges of designing emerging ***billion transistor system-on-chips*** (BTSOCs), it is widely believed that the chips have to be designed at an abstraction level well above RTL. Indeed, system designers typically reason about and conceptualize designs at an abstract functional level where system outputs are described as algorithms or transfer functions of the system inputs. This design abstraction, called ***electronic system level*** (ESL), enables ease of design capture and early design space exploration of multiple design implementation alternatives. ESL designs can be refined into lower levels of abstraction through a number of steps that gradually map abstract functions into ***register-transfer level*** (RTL) components, which is the next level of design abstraction. In this section, we discuss the main drivers and the basic elements of the emerging ESL design method.

5.1.1 ESL design methodology

Moore's law, which states that chip complexity doubles every 18 months, has been the key driver behind the paradigm shift in EDA methodology. Although RTL design methods are dominant currently, rapid growth in chip complexity coupled with shrinking time-to-market windows result in RTL design methods not being able to scale with the complexity of emerging designs. This trend

necessitates fundamental changes that force the move toward higher levels of abstraction. Let us examine two such fundamental changes.

The first fundamental change is that the cost of a new chip design by use of an RTL design method is no longer economically viable.

Example 5.1 Consider a startup company designing a new chip at 65-nm technology. The average design cost with the RTL design method is $30 million. Assuming a fivefold return on investment, the company has to make at least $150 million in sales. Assuming a 10% market share (a respectable goal for a startup), the chip design has to target a $1.5 billion market. The reality is that few such markets exist, and if they do, they would be very crowded.

To dramatically reduce the design cost, the basic building blocks have to be one order of magnitude larger than what is used in RTL. It has been suggested that the basic component becomes a design block with 10,000 to 50,000 gates. The 50,000 gate limit is set so that contemporary RTL-to-GDSII tools can comfortably handle each block without running into issues relating to design complexity explosion that lead to excessive memory requirements and long run times. Note that the complexity of these new building blocks coincides with the complexity of an embedded processor. Indeed, many view processors to be the basic building blocks (*i.e.*, the "gates") of an ESL method.

Example 5.2 Figure 5.1 shows a prediction by the **international technology roadmap for semiconductors** (ITRS) as an implication of Moore's law [SIA 2007]. Assume a constant

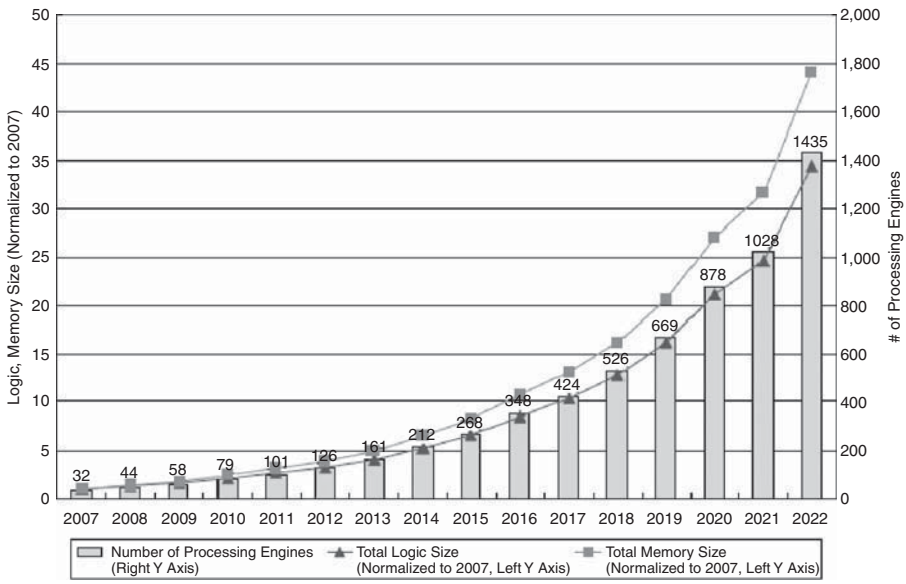


FIGURE 5.1

ITRS 2007 prediction of processing engine count.

die size of 66 mm^2 and that the sizes of the main processor and peripherals remain unchanged. The number of **processing engines** (PEs)—*i.e.*, processors that perform fixed functions—will grow from 32 in 2007 to 79 in 2010 and 268 in 2015. Conceivably, chip designs with below a thousand components are easier to conceptualize and implement.

The second fundamental change is the rapid growth in chip complexity that enables integration of previously separate components on a printed circuit board (*e.g.*, CPU, Ethernet controller, or memory) into a single chip. This integration of heterogeneous "content" comprises not only hardware but also software. Indeed, a chip is more often designed as a programmable computer system, requiring not only hardware but also firmware, operating systems, and application software, with applications often downloaded after the end product is deployed to consumers.

Example 5.3 Figure 5.2 shows a block diagram of Texas Instrument's OMAP platform designed for the cellular phone market. On this chip, one can find a mixture of heterogeneous components, including an ARM programmable processor, an image signal processor, numerous hardware processing engines for acceleration, and peripherals that interface with the outside world.

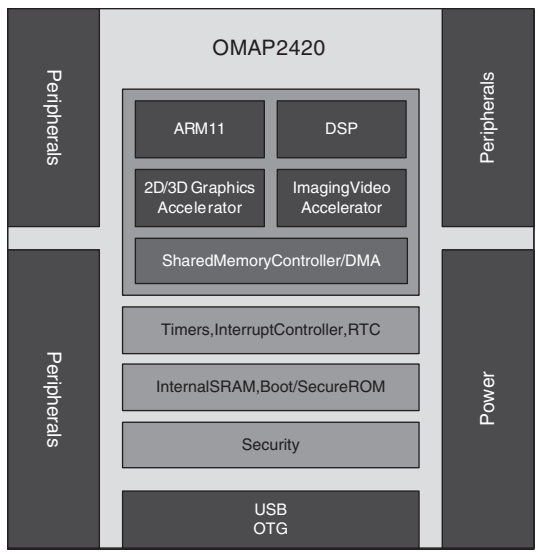


FIGURE 5.2

Texas Instrument's OMAP platform.

To better understand emerging ESL methods, we examine three necessary elements of a design abstraction that allow a chip design to be conceptualized as a connected set of components:

- *Computation*: How to specify what each component does?
- *Composition*: How to assemble different components and coordinate their computation to form a larger system?
- *Communication*: How do the components exchange information?

It is instructive to first examine the RTL design abstraction against these three elements, and consider the design method we build around the RTL abstraction.

Example 5.4 A component in an RTL abstraction can be captured with objects in a **hardware description language** (HDL) (*e.g.*, a module in Verilog, or an entity in VHDL). The computation of a component is captured by its per-cycle behavior, specifying the transformation of register values at each clock cycle. The components can be composed together into a larger design by connecting the ports of components with wires (*e.g.*, with the port map construct in Verilog or VHDL). The communication between components is effected by the transfer of values through wires. With the three design abstraction elements clearly defined, an RTL method can be constructed: RTL synthesis tools convert each module into a gate level design according to its per-cycle behavior, and all modules are stitched together with wires, after which gate level optimizations are performed; RTL simulation tools convert each module into a concurrent software process triggered by events such as the rising edge of a clock.

The ESL abstraction typically conceptualizes the computation of a component at a more abstract level by use of *untimed* behavior. In other words, chip designers do not make the decision as to how computation is mapped to a particular clock step. Currently C/C++ and Matlab are among the most commonly used languages for capturing system behavior. However, the mechanisms for composition and communication differ widely on the basis of the semantics of each specification language.

Next, we examine two ESL methods that are becoming increasingly accepted. For each method, we examine the design abstraction used for composition and communication. We also examine how we map each component to RTL (called *component synthesis*), how we map the full system into RTL (called *system synthesis*), and how we verify the full system.

5.1.2 Function-based ESL methodology

The function-based ESL design method uses a *computational model* [Lee 1996] to compose different functional components into a complete system. The computational model determines how the components execute in parallel and how they communicate with each other. Note that the manner in which components exchange information is also more abstract than RTL.

Example 5.5 A process network [Kahn 1974] is a design abstraction in which functional components, called processes, communicate through data items called tokens. On each execution of a process, input tokens are consumed, and output tokens are produced. A process is executed whenever its input tokens are available. Compared with RTL, all three elements of computation, composition, and communication are more abstract: the computation of each process is an untimed algorithm; the execution of processes follow a partial order; and the communication between processes are through unbounded **first-in first-outs** (FIFOs).

A plethora of computational models has been developed in the past. Some are developed as special cases of a general model. For example, synchronous data flow [Lee 1987] is a special case of process networks that imposes the constraint in which the number of input and output tokens consumed and produced by each process is statically determined to be constant. The synchronous data flow model is useful to capture multi-rate signal processing systems. Other computational models are designed for particular application domains. For example, synchronous models [Halbwachs 1991; Berry 1992], which capture a system as atomic actions in response to external events, are very expressive in capturing control-dominated applications.

In practice, the most widely used computational models arise from models used for the simulation of dynamic physical systems. This is not surprising: HDLs at the RTL abstraction (*e.g.*, Verilog/VHDL) were in fact languages originally used for the simulation of digital systems. One of the most widely used ESL computation models in the industry is Simulink developed by MathWorks. Like a process network, Simulink graphically captures a system as a connected network of components, called a *block diagram*. Here each block captures the instantaneous behavior of a component, in other words, how the component output changes given component input and state variable. The connections between components, or *signals*, serve as constraints, subject to which the entire dynamic system can be solved to find the relationship between signals and state variables over a time period. As a result, Simulink can be used to simulate both continuous time systems (often analog subsystems) and discrete time systems (often digital subsystems). A certain execution order of the blocks is imposed in each iteration of system solving. For example, if block A's output drives the input of another block B, it is required, according to the execution semantics of Simulink block diagram, that A is executed before B. This execution order coincides with process network. Therefore, Simulink can serve as an executable modeling environment for process network and other computational models.

Verification in a function-based method is achieved through simulation. A simulator respecting the underlying computational model is typically used.

In a function-based design method, component synthesis can be achieved through a number of paths:

1. Direct translation to RTL
2. Direct mapping to predesigned intellectual property component

3. High-level synthesis to RTL
4. Compilation to software programs

The most widely used forms of system component synthesis are to directly map each component into separate hardware or map all components into software running on a single processor. In the former case, point-to-point communication is implemented by hardware FIFOs or ping-pong memories.[1] In the latter case, communication is trivially implemented by shared memory.

Example 5.6 Consider the **digital signal processor** (DSP) builder product from Altera. It uses Simulink as the design specification and simulation environment. In addition, it supplies a large library of predesigned, configurable intellectual property (IP) components, each with both a Simulink simulation model, and an RTL implementation. When a DSP designer uses Simulink to design a system with the predesigned components, DSP builder can automatically generate a full-system RTL by mapping each component to its corresponding IP component, as well as the top level interconnection. Xilinx' SystemGenerator product and Berkeley's Chip-In-A-Day, share a similar methodology.

5.1.3 Architecture-based ESL methodology

The architecture-based ESL method follows closely the traditional discipline of computer organization. Here a design is conceptualized as a set of components, including processors, peripherals, and memories. Because these components are often available as reused designs, either from previous projects or acquired from third parties, they are referred to in the industry as *intellectual property* components, or simply IPs. The components are connected through buses, switch fabric, or point-to-point connections. Components connected to buses and switches typically contain a set of master and/or slave ports. Each slave port is assigned an address range so that targeted communication can be identified. Several bus-based communication protocols are commonly used for this purpose, including AMBA from ARM and CoreConnect from IBM. Furthermore, the industry-wide SPIRIT consortium has developed IP-XACT, a standard to help specify the composition of architecture-based systems with IP blocks.

A key difference in the architecture-based method is that a processor component is not required to implement a fixed function. Instead, the computation of the processor can be "programmed." Such programming can happen in the traditional sense in which the component is a programmable processor and is programmed post-silicon. Or it can happen in the design exploration phase in which an accelerator is assigned a certain function to be synthesized into hardware. In this context, it is important for a processor to present a *programming*

[1] Ping-pong memories are two SRAMs alternately accessed by a producer and a consumer component. By switching the access of the SRAMs, data can be exchanged between the producer and the consumer without the need for copying data.

model to facilitate software development. For example, a RISC processor defines an instruction set into which application software can be compiled. We use the term *program* broadly to refer to system software, application software, as well as the accelerator functions.

Communication in the architecture-based method is abstracted as a set of *transactions*. Typically, a transaction is either a bus transaction, which represents a (burst) *read* or *write* operation to peripherals or memories, or a point-to-point data transfer. **Transaction level modeling** is a popular modeling style in which a construct called *channel* is used to encapsulate or abstract away the concrete protocols for such transactions [Zhu 1997]. The use of abstract channels allows faster high-level design exploration, because unnecessary details are abstracted away. In addition, it significantly eases the task of verification because testbenches can be written at a high abstraction level. The abstract channel can be replaced easily by a *transactor*, which encapsulates the timed protocol, whenever a detailed simulation or implementation is required.

In the architecture-based method, the system architecture and its program are simulated together. Two popular approaches for simulation exist currently. The *SystemC–based approach* [Grötker 2002] models the architecture and program together in a single environment. Here the program for a processor is directly modeled as a C++ class by extending a predefined SystemC class. The communication ports (an architectural element) of the processor are directly accessible to the program. On the other hand, the *virtual prototyping approach* has a clear separation between the architecture and the program it runs. Here a system is first constructed by connecting a set of virtual components predesigned for emulating hardware devices. Such virtual components are referred to in the industry as verification IPs. Together, they present a *programmer's view* or a programming model. The program is typically captured as a binary executable and can be loaded for simulation. The SystemC approach is suitable for the design phase when the system architecture is not yet well defined and requires extensive exploration. Virtual prototyping is more suitable for use when the system architecture is relatively well defined, enabling concurrent development of the software and the hardware.

In the architecture-based design method, component (IP) synthesis can be achieved by the following approaches:

1. Direct instantiation of a predesigned IP component.
2. Extension of the processor design with processor configuration and instruction set extension.
3. Synthesis of the processor from an ***architecture description language*** (ADL) specification [Mishra 2008].
4. High-level synthesis to RTL.

Approach 1 requires the least involvement with users. However, mechanisms have to be established to ensure that the simulation model and the instantiated RTL model are consistent with each other. Approaches 2 and 3 are used for

application-specific instruction processors, with varying degrees of user freedom in defining the processor instruction set and microarchitecture. Typically, suppliers of such components create the compiler and simulator tool chain according to the instruction set definition or extension. Approach 4 is used when a hardwired accelerator has to be created to meet exacting performance, power/energy, or cost constraints, when an existing IP cannot satisfy such constraints.

In the architecture-based design method, system synthesis mainly involves the synthesis of system-level interconnect and the generation of the top level design. Such a method is often referred to as a *communication-centric design method*. Many ***computer-aided design*** (CAD) vendors offer *IP assembly* tools to help assemble these architectural components into a computer system. For instance, given a system description in the form of IP-XACT, tools such as Synopsys *coreAssembler* and Mentor Graphics *Platform Express* can be used to generate the top level netlist, as well as the interconnect fabric. Xilinx's *EDK*, and Altera's *SOPC Builder*, fill the same role for ***field programmable gate arrays*** (FPGAs). The generated fabric typically respects an on-chip bus protocol standard. One of the most widely used bus standards is ARM's AMBA/AXI bus. Commercial products are available to generate specialized circuits for on-chip buses conforming to the AMBA bus standards.

5.1.4 Function architecture codesign methodology

A more ambitious form of ESL design methodology is function architecture codesign. As shown in Figure 5.3, this method follows a top-down, stepwise refinement approach in which designers start with design requirements followed by the development of a functional model. As in the case of the function-based ESL method, here the functional model consists of a network of functional components under a specific computational model, thereby capturing the system function as a relation between system output and input. This functional model is gradually refined into an architectural model, which as in the case of architecture-based ESL method, consists of a network of architectural components that communicate at the transaction level. The architectural model can be further refined into RTL by a step of high-level synthesis as described in the next section. Finally, the RTL model can be implemented in silicon by means of "*RTL-handoff*" to the ensuing steps of logic synthesis and physical design.

An important aspect of this method is verifiability. In an ideal function architecture codesign method, a common executable modeling language is used for both functional and architectural modeling. This ensures that at every refinement step, an executable model is created that can be simulated to confirm correctness or to collect performance metrics. Thus, this method is particularly useful for system architects to perform architecture exploration. A pioneering modeling language to support such methodology is SpecC developed at University of California at Irvine [Gajski 2000]. A pioneering commercial environment

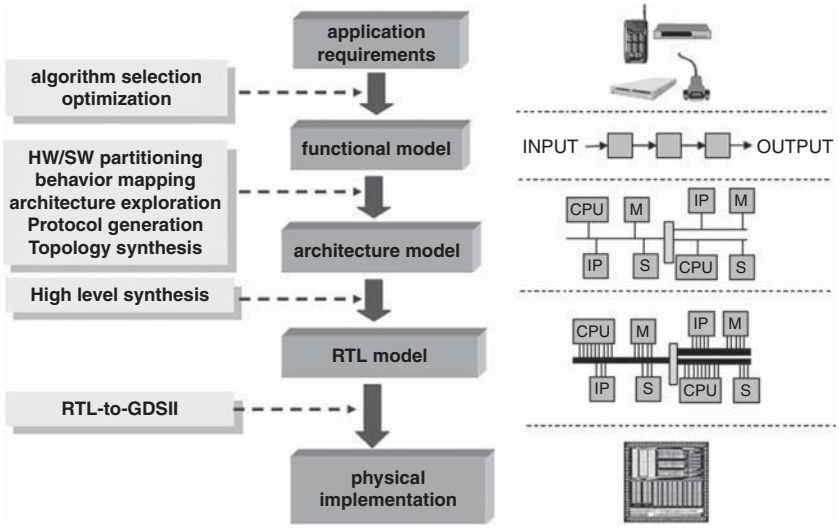


FIGURE 5.3

Function architecture codesign method.

is the VCC tool developed in Cadence [Krolikoski 1999]. Today, SystemC-based environments are widely used for this purpose.

Although top-down function architecture codesign is one of the earliest ESL methods advocated by several researchers, its deployment as a commercially viable method has not yet been realized. The reasons are twofold. First, many ***system-on-chips*** (SOCs) are typically designed as a platform that can be extended with many derivatives to serve many applications. The idealized top-down approach, which ties the architecture to a particular function or application, makes it difficult to extend the platform for derivatives. In this case, the architecture-based method is more practical. Second, when a system is captured in a domain-specific computational model, it is often the case that the best architecture template is already known, and the architecture instance is best generated by automation rather than through manual refinement. In this case, the functional-based method may be more practical. However, in general, a more practical meet-in-the-middle method (that combines the top-down and bottom-up approaches) may be best suited for a number of applications.

5.1.5 High-level synthesis within an ESL design methodology

We now examine the role of high-level synthesis within an ESL design method. High-level synthesis is an automated method of creating RTL designs from

algorithmic descriptions. Within an ESL design method flow, we consider the following usage models of high-level synthesis:

1. Functional component synthesis
2. Co-processor synthesis
3. Application processor synthesis

Functional component synthesis is used in a function-based ESL method. Because the communication semantics are well defined under a computation model, high-level synthesis can be used to create the internal design of each component, while respecting the communication constraints at the inputs and outputs of each component. This enables the construction of larger systems. For example, in a process network–based ESL method, each component has a set of FIFO ports for communication. Therefore, each component can be synthesized to run asynchronously with respect to other components.

Coprocessor synthesis is used in an architecture-based ESL method when part of an application is implemented as software running on a programmable processor and part of the application is implemented as a hardware accelerator. In this case, *hardware/software partitioning*, be it manual or automated, has to be performed to decide on the division of responsibility. Two criteria are typically used to make partitioning decisions: performance/power and flexibility. It is preferable to implement performance/power critical portions of the application in hardware, and it is preferable to implement those that require post silicon changes into software. Once the partitioning is performed, high-level synthesis is used to create the accelerator, and interface synthesis is used to create the software/hardware communications.

Example 5.7 Consider the implementation of an MPEG video encoder. The algorithm divides a video stream into frames, and each frame is divided into slices, where each slice contains a sequence of macro blocks, which are 8×8 pixels. Although most of the algorithm deals with management and configuration, the profiling result shows that much of the program run time is spent on processing each macro block: including stages such as motion estimation (ME), discrete cosine transform, Huffman encoding, and run length encoding. Consequently, a typical hardware/software partitioner would synthesize the processing pipeline (with high-level synthesis) into hardware accelerators, whereas the rest of the design is implemented as software.

In many occasions where performance or power is more important than post silicon programmability, application processor synthesis is used in an architecture-based method to synthesize the entire, stand-alone program into custom hardware. Because in an architecture-based method, component processors communicate through transaction-level ports (such as streaming or bus ports), the mapping from a C programming model to interface hardware is well defined. RTL synthesized by high-level synthesis can be considered as "drop-in"

replacement of programmable processors. Large systems can thus be constructed with the well-established IP assembly method.

High-level synthesis has seen intensive research in academia since the 1980s, and many believed that it would succeed RTL synthesis as the dominant design method. Unfortunately, this transition did not happen for a number of reasons. One factor limiting the commercial success of high-level synthesis is its lack of scalability: although large RTL designs can be constructed by composing smaller RTL designs, it is not as easy to compose designs created by high-level synthesis together with legacy RTL designs. Without a well-defined system-level design method, there is no standard way of defining how algorithmic components communicate with each other. Given today's complex heterogeneous systems-on-chip, high-level synthesis in practice has to become a component synthesis tool in the context of an ESL method as opposed to a full chip synthesis method originally envisioned in the early days of HLS research.

Furthermore, both functional and architecture-based ESL methods have matured sufficiently that the standard design flow is converging in the industry. Because the design of new hardware logic is often where it costs chip companies labor and where they achieve product differentiation, it has become increasingly important to add high-level synthesis to the emerging design flow.

5.2 FUNDAMENTALS OF HIGH-LEVEL SYNTHESIS

In a nutshell, ***high-level synthesis*** (HLS) takes as input an algorithmic description (*e.g.,* in C/C++) and generates as output a hardware implementation of a microarchitecture (*e.g.,* in VHDL/Verilog). Algorithmic languages such as C/C++ capture what we refer to as the *behavioral-level* (or high-level) description of the design; whereas hardware description languages such as VHDL/Verilog capture the ***register-transfer level*** (RTL) description of the design.

Figure 5.4a shows a typical example of an input behavioral level description. Here a design is described as a sequence of statements and expressions operating on program variables. Such description captures the function of the design without any hardware implementation detail.

Figure 5.4b shows a typical output of HLS as an RTL description in a form known as ***finite state machine with datapath*** (FSMD) [Gajski 1992]. The FSM controller sequences the design through states of the machine by following the flow of control in the algorithmic behavior, whereas the datapath performs computations on the abstract data types specified in the behavior. The datapath contains a set of registers, functional units, and multiplexers connecting the output of registers to the inputs of functional units, and *vice versa*. The controller takes as input a set of status signals from the datapath and outputs a set of control signals to the datapath. The control signals include those that control the datapath multiplexers, the loading of datapath registers, and the opcode used to select different functions in a functional unit. The state, that is, the control

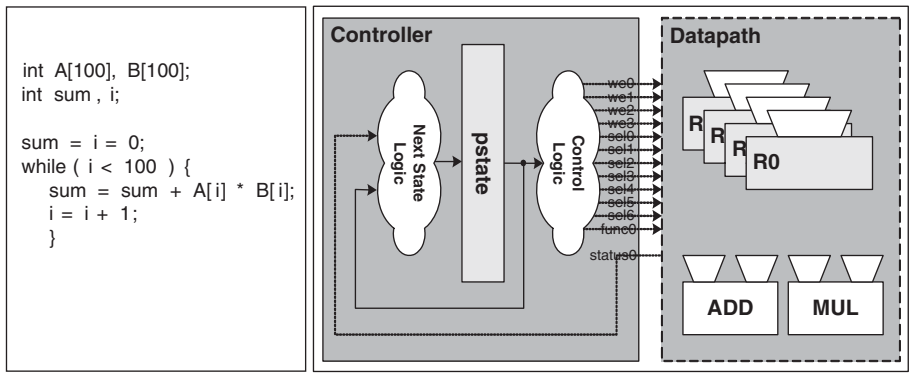


FIGURE 5.4

High-level synthesis input/output.

step a circuit is currently in, is remembered in the controller with a set of flip-flops. Thus, the datapath performs *register transfers*, or computations, that transform values retrieved from some registers and stores the results to other registers; whereas the controller determines "when" certain register transfers are executed by specific valuation of control signals.

HLS effectively transforms an untimed behavioral specification into a clocked RTL design, resulting in a substantial semantic gap between the RTL description and the behavioral description. To bridge this gap, HLS involves a complex sequence of design steps that gradually refines the design behavior into an FSMD design. As shown in Figure 5.5, the structure of an HLS tool naturally resembles a software compiler: it includes a set of *compilation components*. Each component transforms one *program form* to another more detailed program form. We distinguish between two types of program forms: we use *code* to refer to the textual human-readable form, and *representation* to refer to the in-memory machine-readable form (*e.g.*, data structures such as graphs).

The *front-end* component performs the lexical and syntactical analysis of the behavioral program code to build an ***intermediate representation*** (IR) in memory. The IR can be considered as a sequence of operations transforming values retrieved from memory or generated as the result of other operations. The operations typically correspond to arithmetic (addition, subtraction, etc.) or logic computations (AND, OR, etc.).

The *optimizer* component performs program analysis on the IR to extract useful information about the program and transforms the IR to a semantically equivalent but improved IR. Virtually all code optimization algorithms found in software compilers can be applied here, although their effects may differ [Gupta 2003]. For example, common subexpression elimination and dead code elimination can be used to remove redundant code. Tree height reduction, strength reduction, and algebraic

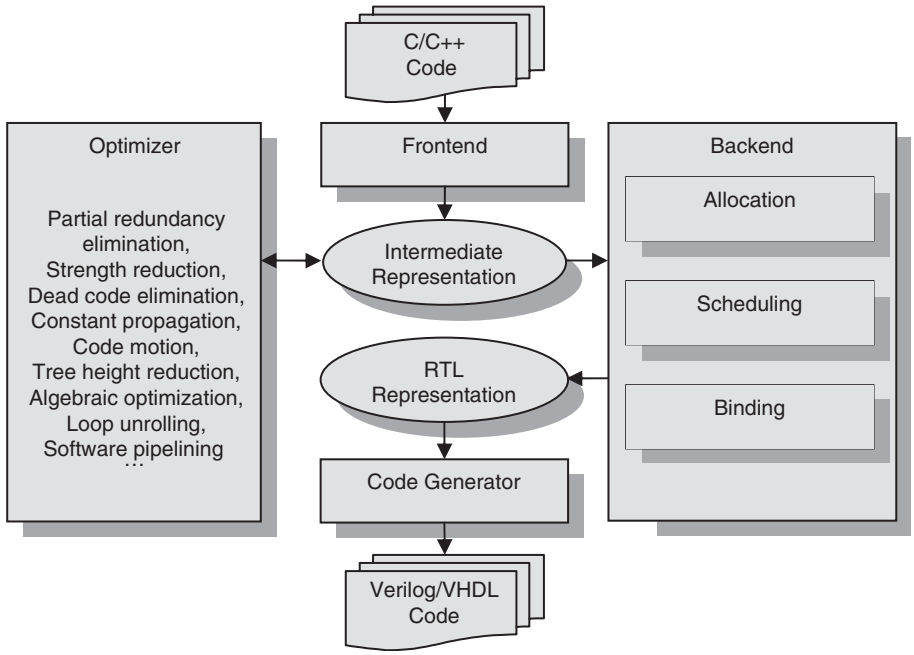


FIGURE 5.5

Typical high-level synthesis flow.

transformations can be used to simplify or speed up the evaluation of expressions. Loop unrolling and software pipelining can significantly improve code in loops.

The core of high-level synthesis resides in the backend, which includes several critical steps to transform the IR into an *RTL representation*. To facilitate the generation of an FSMD in the form of Verilog/VHDL, which performs computation one clock cycle at a time, the RTL representation has to capture more detailed implementation information than initially available in the IR. These critical steps include allocation, scheduling, and binding.

The *allocation* step determines the hardware resources required to implement the operations within the IR to satisfy certain performance requirements. The resources required include storage resources such as registers and functional units such as adders and multipliers. Often, a library of such resources, called modules, is created and characterized in advance for a specific standard cell library. Then allocation involves the choice of which module to use, among potentially many alternatives with differing area and timing, and how many of them.

The *scheduling* step maps each operation in the IR into a particular clock cycle, called a control step. Note that multiple operations can be mapped to the same control step, in which case they run in parallel. In fact, the key objective of scheduling is to find and exploit the parallelism in the sequential code

represented by the IR. The extent to which operations can run in parallel is limited by the dependency among them. For example, if operation A uses the result of operation B, then A can only be executed in later steps than B. It is not always easy to exactly determine the dependency relationship, especially when indirect memory references are involved. In addition, the availability of resources (determined in the allocation step) also constrains what can be scheduled in parallel.

The *binding* step maps each operation to a functional unit (functional unit binding) and the value it computes into a storage resource (storage binding). The purpose of binding is to recognize the fact that not all operations and values require the use of hardware resources at the same time. The nonoverlapping use can be exploited for hardware sharing, thereby minimizing the hardware circuitry demand.

It is important to note that each of the allocation, scheduling, and binding steps involves the solution of a complex optimization problem. These problems can all be elegantly formulated as mathematical optimization problems solving for some decision variables to minimize an objective subject to certain constraints. There are many challenges in solving these problems. First, abstracting each problem mathematically inevitably introduces approximations. For example, the objective to optimize area through the minimization of functional units is often an indirect measure of the true objective designers' concern about reducing silicon area, because this includes not only functional units but also the area of interconnects. Second, it has been shown that all allocation, scheduling and binding problems are NP hard problems. Therefore, one often devises heuristics-based algorithms for practical solution of large-scale problems. Third, the allocation, scheduling, and binding tasks are tightly interdependent. In fact, the preceding three-phase approach is not necessarily the best. Often, one faces the *phase-ordering* problem, where the optimal solution of one phase leads to the suboptimal solution of the other phase. It is thus often necessary to have an iterative improvement solution where these tasks are applied multiple times.

Finally, the *code generator* generates the Verilog/VHDL code that is ready for RTL and logic synthesis. The output code is in FSMD form. The datapath can be derived directly from the allocation and binding decisions. The controller FSM can be derived from scheduling and binding.

Without HLS, designers have to manually generate the RTL design, a painstaking process that requires the manual tasks of scheduling, allocation, and binding. In contrast, HLS automates the task of RTL design, allowing designers to more productively focus their design activities at the behavioral level. Often, there is an order of magnitude difference between the size of the RTL description and the corresponding behavioral description. In addition, the automatically generated RTL can be guaranteed to be correct by construction. This design automation process results in a significant reduction of RTL development and verification effort, yielding a large gain in design productivity.

In the sequel, we use an example to illustrate the process of transforming a behavioral code into Verilog code. For now, we treat compilation components as black boxes and leave the discussion until Section 5.3, while focusing on the series of program forms that are generated at each stage. To enhance understanding and allow for a hands-on treatment of HLS that will permit construction of simple HLS tasks/components, we present simple but completed program forms, called TinyC, TinyIR, and TinyRTL; these can be used as vehicles to construct software implementations of high-level synthesis tools. We begin by defining the three exemplar program forms.

5.2.1 TinyC as an example for behavioral descriptions

The behavioral description of a design is typically captured by a program in a procedural programming language, also known as an imperative programming language. An imperative program captures the computation as a sequence of actions on program state. The program state is defined by the set of all program *variables*. The actions are *statements* that change program states by updating the values of one or many variables. Because this style of programming prescribes the steps to solve a problem, an imperative program is often called an *algorithmic description*.

We use an instruction language, TinyC, throughout this text. TinyC resembles the commonly used C language, but a number of simplifying assumptions are made to facilitate the discussion, as follows:

- All statements are captured in a single procedure, and there are no procedural calls.
- Only 32-bit signed integer (int) and Boolean (bool) primitive types are supported.
- Only one-dimensional arrays are supported for aggregate types.
- No pointers are supported, as a result, the address variables cannot be taken, and no memory is allocated dynamically.

Although TinyC is a fully functional language that captures the essence of imperative languages, it is important to recognize that the excluded features present in the real-world languages make the job of high-level synthesis substantially more difficult. We will deal with this subject in Section 5.4.

Example 5.8 Figure 5.4a shows the dot product algorithm in TinyC. Dot product is widely used in engineering. The dot product of two vectors, A and B, is defined to be the sum of products of their respective elements. In TinyC, the two vectors, A and B, are declared as arrays. Two scalar variables, sum and i are declared. A while-loop is used to calculate the result, in which each loop iteration is used to calculate the partial sum up until the i^{th} iteration.

The language definition of TinyC can be defined with a set of production rules in ***Backus-Naur Form*** (BNF). Each production rule defines a language construct, called a non-terminal, by the composition of other non-terminals,

or terminals. Like a word in English, terminals are atomic units of a language. The non-terminals can be recursively defined.

Definition 5.1 The syntax of TinyC is defined as follows, which includes the following constructs:

- ***Declarations****, which define scalars or array variables*
- ***Statements****, which are either assignment statements, leading to a change of program state, or control flow statements*
- ***Expressions****, which are either transformations of scalar values using predefined operators, or defined primitive values, such as integer or Boolean literals, or program variable values.*

```
program:
    declaration* statement*

statement:
    variable '=' expression ';',
    'if' '(' expression ')' statement
            ( 'else'  statement )* ,
    'while' '(' expression ')' statement ,
    'break' ';',
    '{' declaration* statement* '}'

declaration:
    type  identifier ['=' expression]';',
    type  identifier '[' expression ']' ';'

type:
    'int' , 'bool'
```

```
expression:
    '-' expression , '!' expression ,
    expression '+' expression ,
    expression '-' expression ,
    expression '*' expression ,
    expression '/' expression ,
    expression '^' expression ,
    expression '>>' expression ,
    expression '<<' expression ,
    expression '&' expression ,
    expression '|' expression ,
    expression '='  expression ,
    expression '!=' expression ,
    expression '<' expression ,
    expression '<=' expression ,
    expression '>' expression ,
    expression '>='  expression ,
    '(' expression ')'  ,
    integer , identifier , 'TRUE'  ,  'FALSE'  ,
    identifier '[' expression ']'
```

The language can be used to construct a parser, which parses the textual program into a data structure suitable for machine manipulation. In the case of HLS, the parser generates an intermediate representation (as represented by the front-end component in Figure 5.5).

5.2.2 Intermediate representation in TinyIR

The purpose of the intermediate representation (IR) is to separate the optimization algorithms from input languages and target architectures.

Before we discuss the IR in detail, we need a notation to capture an IR. In practice, an IR is implemented in software as a complex data structure. However, presenting the IR in this form introduces unnecessary implementation details. To be concise and precise, we elect to use a more abstract mathematical notation to describe the IR. In the following, we outline how we can replace data structures with mathematical objects. The readers should do the opposite when they translate the algorithms presented in this chapter into software implementation.

- A data type T corresponds to a **set** T; in particular the integer type Z corresponds to set Z.
- A linked list or arrays whose elements are of type T corresponds to the **power set** of T, or the set of all subsets of T, denoted as $T[]$.
- A record with fields a of type A, and b of type B corresponds to a set of **named tuples**, denoted as $\langle a : A, b : B \rangle$.
- A graph R whose nodes are of type A corresponds to a **relation** $R : A \times A$.
- A hash table or dictionary F that maps a value of type A to a value of type B corresponds to a **function** $F : A \mapsto B$.

This mathematical notation is used later in Section 5.3 to present HLS algorithms, because we can use it to represent complex operations on data structures. For example,

- Use function application $F(a)$ to represent a dictionary lookup.
- Use $a \in A$ to perform a set membership test, and use $\forall a \in A$ to enumerate all members in A; and use $A[i]$ to retrieve the i^{th} element in A.
- Use $a\ R\ b$ to check whether b is a predecessor of a in graph (relation) R.

We are now ready to discuss TinyIR, a simple IR designed to sufficiently capture programs in TinyC.

Definition 5.2 A TinyIR is a tuple $\langle O, S, V, B \rangle$ with the following elements:

- *A set* $O = \{lds, sts, lda, sta, ba, br, cnst, +, -, *, /, <<, >>, \ldots\}$ *of operation codes, which corresponds to the set of all virtual instruction types.*
- *A set* S *of* ***symbols****, which corresponds to the scalar and array variables.*
- *A set* V: $\langle opcode: O, src1: V, src2: V, symb: S \cup B \cup Z \rangle$ *of* ***virtual instructions****, which corresponds to the expressions and control transfers in the program.*
- *A set* B: $V[]$ *of* ***basic blocks****, each containing a sequence of virtual instructions.*

Constructs in TinyC (*e.g.*, declarations, statements, and expressions) have equivalent representation in TinyIR: declarations correspond to symbols, and statements and expressions correspond to virtual instructions. In particular, the control transfer statements are converted into *ba* (branch always) or *br* (branch if true) instructions. Variable accesses are converted to *lds* (load scalar) or *lda* (load array) instructions, and assignments are converted into *sts* (store scalar) or *sta* (store array) instructions. Expressions in TinyC have a one-to-one correspondence to virtual instructions in TinyIR.

Each instruction is a tuple with an *opcode* field; zero, one, or two source operands; and optionally a *symb* field. For branch instructions, the *symb* field defines its branching target, that is, which basic block it branches to. For load/store instructions, the *symb* field defines the symbol corresponding to the scalar or array variable it accesses. For constants, it defines the Boolean or integer constant value.

Although the sequence of virtual instructions completely defines the behavior of the program, in TinyIR the virtual instructions are grouped within different basic blocks. Only the last instruction of a basic block could be a branch

instruction. In the other words, the instructions in a basic block are what we usually referred to as "straight line code."

Example 5.9 In the following we show the dot product program in Example 5.8 in TinyIR form. There are two basic blocks, B1 and B2. Each virtual instruction is uniquely numbered. When the result of an instruction is used as an operand for another instruction, its number is used. Note that B2 is a loop, indicated by the bt instruction (15), which is a branch branching to the beginning of B2.

```
scalar sum;
scalar i;
array  A[100];
array  B[100];

B1:
(0)    cnst 0
(1)    sts (0), sum
(2)    sts (0), i

B2:
(3)    lds i
(4)    lda (3), A
(5)    lda (3), B
(6)    * (4) (5)
(7)    lds sum
(8)    + (6) (7)
(9)    sts (8), sum
(10)   cnst 1
(11)   + (3) (10)
(12)   sts (11), i
(13)   cnst 100
(14)   < (11) (13)
(15)   bt (14), B2
```

5.2.3 **RTL representation in TinyRTL**

An RTL representation is to HLS what assembly is to a software compiler. Like assembly, the RTL representation captures key microarchitectural information. In the case of a software compiler, the processor architecture is predetermined; therefore, the assembly only exposes these microarchitecture features, for example, the architectural registers available, the instruction set, etc. In the case of HLS, the microarchitecture is synthesized; therefore, the RTL representation has to convey what architecture resources are allocated and how they are used.

One key difference between an RTL representation and an IR is that microarchitecture resources are introduced. There are two types of microarchitectural resources: **computational resources**, which are functional units that perform logical, relational, and/or arithmetic functions; and **storage resources**, which include memories and registers. A memory typically corresponds to a on-chip static RAM used to store scalar or array variables, whereas a register is an array of flip-flops used to store scalar or temporary values.

An instruction in RTL representation is referred to as a *register transfer*. A key difference between a register transfer and a virtual instruction in an IR is that the former is annotated with microarchitecture resource use. For example, most register transfers designate a destination register. Likewise, the source operands of a register transfer are registers. In addition, each register transfer designates the functional unit executing the instruction.

The register transfer representation is **cycle accurate** in the sense that the clock cycle (control step) at which a register transfer is executed is fully specified. This level of detail makes it possible to generate a sequential circuit implementation of the program.

Definition 5.3 A TinyRTL is a tuple $\langle M, R, U, I, C \rangle$ with the following elements:

- A set M of **memories** used to store scalar and array variables.
- A set R of **registers** used to store scalar variables or temporary instruction results.
- A set U of **functional units**, such as adders, subtractors, multipliers, shifters, etc.
- A set $I : \langle unit : U, opcode : O, dest : R, src1 : R \cup S \cup Z \cup C, src2 : R \rangle$ *of* **register transfers**, each of which uses a functional unit to transform values, which are either constants or retrieved from registers, and stores the result back to a register.
- A set *C: I[]* of **control steps**, each of which contains a set of register transfers.

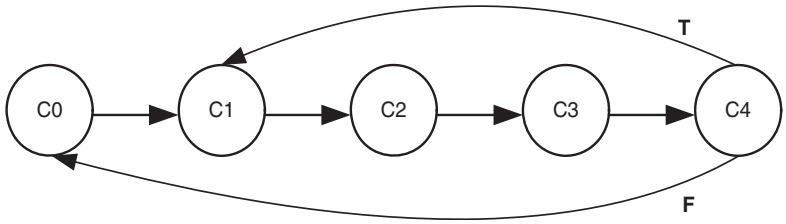
Example 5.10 The dot product algorithm of Example 5.8 is shown in TinyRTL form. Here C0-C4 corresponds to the set of control steps. Each control step contains one or more register transfers. It is instructive to compare the TinyIR form and TinyRTL form. Note that most virtual instructions in TinyIR are translated into register transfers. Some virtual instructions (e.g., constants) degenerate into direct operands, because it takes nothing to compute them. Also note that almost all register transfers are annotated with the computational resources they use, with the exception of scalar store (in C0) and branch instructions (in C4), because they involve simply copying values to registers but not computation.

```
register R0, R1, R2, R3;
memory M;
unit U0, U1;

C0: sts 0, R0; sts 0, R1;
C1: M.lda R2, R1, A;
C2: M.lda R3, R1, B; U0.+R1, R1, 1;
C3: U1.*R2, R2, R3; U0.<R3, R1, 100;
C4: U0.+R0, R0, R2; bt C1, R3;
```

5.2.4 Structured hardware description in FSMD

Given an RTL representation in memory, we are ready to produce Verilog/VHDL code to drive the downstream RTL-to-GDSII design flow. The key task of the code generator is thus to convert the RTL representation, still in a functional form, to a structural form, that is, as a connected network of components. We use FSMD as a template and generate the controller and datapath separately.

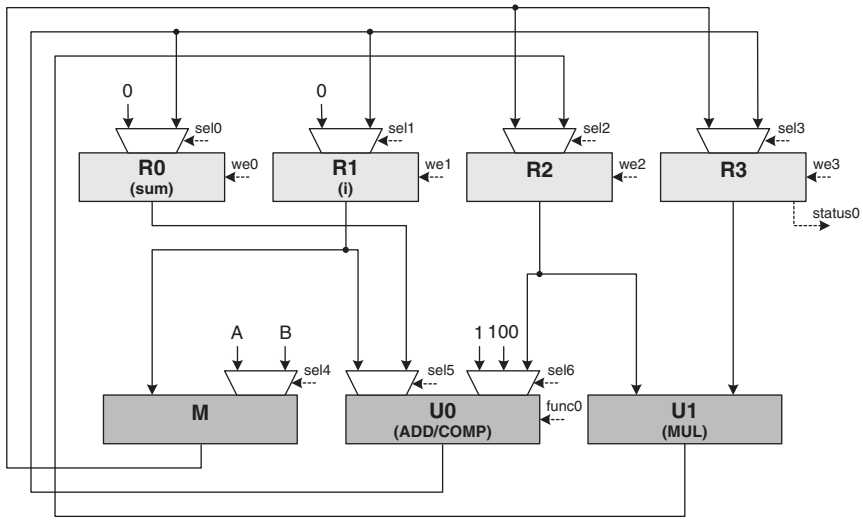
Example 5.11 The FSM diagram and Verilog code for the controller are shown in the following. Note the correspondence between the control steps in the RTL representation and the states in the FSM. Also note the correspondence between the register transfers in each control step and the control signal valuations.




```
`define C0  3'b000
`define C1  3'b001
`define C2  3'b010
`define C3  3'b011
`define C4  3'b100

module ctrl(
  clk, rst, status0,
  we0, we1, we2, we3,
  sel0, sel1, sel2, sel3, sel4, sel5, sel6,
  func0
  );

  input        clk, rst;
  input        status0;
  output       we0, we1, we2, we3;
  output       sel0, sel1, sel2, sel3, sel4, sel5;
  output [1:0] sel6;
  output       func0;

  reg    [2:0] pstate, nstate;
  reg          we0, we1, we2, we3;
  reg          sel0, sel1, sel2, sel3, sel4, sel5;
  reg    [1:0] sel6;
  reg          func0;

  // present state register
  always@( posedge clk or negedge rst )
    if( !rst )
      pstate <= `C0;
    else
      pstate <= nstate;

  // next state logic
  always@( pstate or status0 )
    case( pstate )
      `C0: nstate = `C1;
      `C1: nstate = `C2;
      `C2: nstate = `C3;
      `C3: nstate = `C4;
      `C4: if( status0 )
             nstate = `C1;
           else
             nstate = `C0;
      default: nstate = `C0;
    endcase
```

```
  // control signals
  always@( pstate ) begin
    we0 = 1'b0; we1 = 1'b0;
    we2 = 1'b0; we3 = 1'b0;
    sel0 = 1'bx; sel1 = 1'bx;
    sel2 = 1'bx; sel3 = 1'bx;
    sel4 = 1'bx; sel5 = 1'bx;
    sel6 = 2'bxx; func0 = 1'bx;
    case( pstate )
      `C0:begin
          we0 = 1'b1; we1 = 1'b1; sel0 = 1'b0;
          sel1 = 1'b0;
        end
      `C1:begin
          we2 = 1'b1; sel2 = 1'b0; sel4 = 1'b0;
        end
      `C2:begin
          we1 = 1'b1; we3 = 1'b1; sel1 = 1'b1;
          sel3 = 1'b0; sel4 = 1'b1; sel5 = 1'b0;
          sel6 = 2'b00; func0 = 1'b0;
        end
      `C3:begin
          we2 = 1'b1; we3 = 1'b1; sel2 = 1'b1;
          sel3 = 1'b1; sel5 = 1'b0; sel6 = 2'b01;
          func0 = 1'b1;
        end
      `C4:begin
          we0 = 1'b1; sel0 = 1'b1; sel5 = 1'b1;
          sel6 = 2'b10; func0=1'b0;
        end
    endcase
  end

endmodule
```

Example 5.12 The datapath of the dot product example is shown in the following as a schematic diagram and its corresponding Verilog code. Note that each r $\in$ *R and u* $\in$ U is mapped directly to a hardware resource. Multiplexers are inserted at the input of each register and inputs of each functional unit. A multiplexer degenerates into a wire if it has only one input. For example, both inputs of the multiplier connect only to R2 and R3, respectively; therefore, there is no need for multiplexers.




```
`define A 8'd0
`define B 8'd100

module datapath(
  clk, rst,
  we0, we1, we2, we3,
  sel0, sel1, sel2, sel3, sel4, sel5, sel6,
  func0, rdata,
  status0, base, offs
  );

  input        clk, rst;
  input        we0, we1, we2, we3;
  input        sel0, sel1, sel2, sel3, sel4, sel5;
  input  [1:0] sel6;
  input        func0;
  input  [7:0] rdata;
  output       status0;
  output [7:0] base, offs;

  reg    [7:0] R0, R1, R2, R3;
  reg    [7:0] nR0, nR1, nR2, nR3;
  wire   [7:0] U0, U1;
  reg    [7:0] baseIn, offsIn;
  reg   [7:0] U1In0, U1In1;
  reg   [7:0] U0In0, U0In1;

  // registers
  always@( posedge clk or negedge rst )
    if( !rst ) begin
      R0 <= 8'b0;
      R1 <= 8'b0;
      R2 <= 8'b0;
      R3 <= 8'b0;
    end
    else begin
      if( we0 ) R0 <= nR0;
      if( we1 ) R1 <= nR1;
      if( we2 ) R2 <= nR2;
      if( we3 ) R3 <= nR3;
  end

  // registers' input multiplexers
  always@( sel0 or sel1 or sel2 or sel3 or
           U0 or U1 or rdata ) begin
    case( sel0 )
      1'b0: nR0 = 8'b0;
      1'b1: nR0 = U0;
    endcase
    case( sel1 )
      1'b0: nR1 = 8'b0;
      1'b1: nR1 = U0;
    endcase
    case( sel2 )
      1'b0: nR2 = rdata;
      1'b1: nR2 = U1;
    endcase
    case( sel3 )
      1'b0: nR3 = rdata;
      1'b1: nR3 = U0;
    endcase
  end

  // functional units
  assign U1 = U1In0 * U1In1;
  assign U0 = !func0 ? U0In0 + U0In1 :
                               U0In0 < U0In1;

  // functional units'/memory input multiplexers
  always@( sel4 or sel5 or sel6 or
           R0 or R1 or R2 or R3 ) begin
    baseIn = R1;
    case( sel4 )
      1'b0: offsIn = `A;
      1'b1: offsIn = `B;
    endcase
    case( sel5 )
      1'b0: U0In0 = R1;
      1'b1: U0In0 = R0;
    endcase
    case( sel6 )
      2'b00: U0In0 = 8'd1;
      2'b01: U0In0 = 8'd100;
      2'b10: U0In0 = R2;
    endcase
    U1In0 = R2;
    U1In1 = R3;
  end

  // outputs
  assign status0 = R3[0];
  assign base = baseIn;
  assign offs = offsIn;

endmodule
```

5.2.5 **Quality metrics**

When using high-level synthesis to create RTL hardware for a given application, it is important to, first, satisfy the performance requirements of the application and, second, choose among the best implementation alternatives. To quantify both requirements and quality, we need to establish certain metrics.

Because most applications targeted by high-level synthesis consume and produce large volumes of data, the performance requirement is often dictated by input/output *bandwidth*, defined to be the amount of data consumed/produced per second. The unit of bandwidth varies, depending on the domain of application. For example, triangles per second for 3-D graphics; packets per second for networking; and pixels per second for imaging or video.

The bandwidth requirement can be translated into a performance requirement for the datapath. Typically, the *work*, or the amount of computation an algorithm applies per unit of data, can be measured by the number of instructions required for the computation. Although the precise meaning of an instruction varies depending on the architecture used for implementation, we can use elementary operations, such as those defined by the virtual instruction set of TinyIR, as a rough but implementation-independent measure. Combining bandwidth and work, we can obtain the performance requirement for the datapath in ***millions of instructions per second*** (MIPS).

$$perf(TinyIR) = bandwidth \times work$$

Example 5.13 Consider an Ethernet application in which the wire speed is 4 gigabits per second (raw bandwidth). Assuming a minimum packet size of 64 bytes and a 20-byte preamble between packets, the application needs to process 5.95 million packets per second (bandwidth). Assuming the work is 1000 instructions per packet, then the performance requirement is $5.95 * 1000 = 5950$ MIPS.

We now turn to quality metrics of the RTL implementation. Although it is possible to evaluate the metrics accurately after the Verilog/VHDL is generated and synthesized, these metrics are often too late to be useful for HLS at these downstream stages. We, therefore, need an early, fast, yet reasonably accurate estimation of important metrics. In this section, we develop a "back-of-the-envelope" method that can quickly assess quality metrics directly from the RTL representation.

To calculate the performance of the RTL, we need to estimate the "wall clock" time used for completion of the synthesized algorithm. Assuming the algorithm processes one unit of data, we then have:

$$perf(TinyRTL) = \frac{work}{CycleCount(TinyRTL) * CycleTime(TinyRTL)}$$

Here *CycleCount* is the number of clock cycles it takes to complete the algorithm, whereas *CycleTime* is the shortest clock period for correct operation of the synthesized circuit.

To estimate *CycleCount*(*TinyRTL*), we need to know the number of times each control step is executed. This can be obtained by statically examining the RTL. Alternately, functional simulation can be performed to collect execution count statistics, a process known as *profiling*.

Example 5.14 Consider the dot product algorithm in TinyRTL shown in Example 5.12. It can be shown that C0 will be executed once, whereas C1-C4 will be executed 100 times. Therefore CycleCount(TinyRTL) = 1 + 4 * 100 = 401. Note that this is significantly less than the total number of virtual instructions, which can be calculated from TinyIR as 3 + 13 * 100 = 1303. This type of speedup is achieved by executing multiple instructions in parallel in RTL.

CycleTime(*RTL*) is more difficult to estimate. Recall that the cycle time of a sequential circuit equals the worst-case delay along all register-to-register paths. To calculate such a delay, we first have to establish the delay of individual components along a path.

The delay of a component (in nanoseconds or picoseconds)—if not already available—can be obtained by precharacterizing commonly used components. However, such a method depends on the cell library, as well as the fabrication process. We choose to use fanout four delay (FO4) as the delay unit. FO4 is the delay of a minimal sized inverter driving four identical copies of itself. It has been shown that FO4 delay scales linearly with the feature size (the drawn gate length) of the fabrication process and can be estimated with the following:

$$\text{FO4} = 0.36ns/um * L_{drawn}$$

With FO4 delay, we can express our delay estimation in a process-independent manner and use the preceding formula when we need to find out the absolute delay value.

Example 5.15 The FO4 delay of commonly used 32-bit components is shown in Table 5.1. Under 90nm fabrication technology, FO4 = 0.36ns/um*0.09um = 32.4ps. Therefore, Delay (adder) = 10*32.4ps = 0.324ns. Delay(multiplier) = 35*32.4ps = 1.13ns.

Table 5.1 Component delays

register		functional unit		multiplexer		SRAM	
	FO4		FO4		FO4		FO4
setup	2.0	adder	10.0	2-input	2.4	1KB	10.5
hold	0.0	comparator	6.0	4-input	3.2	4KB	12.0
clock skew/jitter	4.0	multiplier	35.0	8-input	4.8	8KB	12.8
clock to Q	4.0	inverter	1.0	16-input	8.0	64KB	15.0

Without examining the structural representation, we attempt to estimate the cycle time from TinyRTL representation. We can find all *true* register-to-register paths by examining each register transfer in TinyRTL.

$$CycleTime(TinyRTL) = MAX_{rt \in T} Delay(rt)$$

We now consider the delay of a register transfer $rt = \langle u,op,dest,src1,src2 \rangle$. Let us denote the state register of controller to be *pstate* and the multiplexers for the two operands of the functional unit and the destination register to be *mux1*, *mux2*, and *muxd,* respectively. Recall that in an FSMD model, the register-to-register delays involve paths within the datapath, to the FSM controller, and between the datapath and the FSM controller. Accordingly, we have the following paths to consider:

- Path 1: src1→mux1→u→muxd→dest
- Path 2: src2→mux2→u→muxd→dest
- Path 3: pstate→c1→mux1→u→muxd→dest
- Path 4: pstate→c2→mux2→u→muxd→dest
- Path 5: pstate→cu→u→muxd→dest
- Path 6: pstate→cd→muxd→dest

Paths 1 and 2 are from source operand to destination register in the datapath. Paths 3 and 4 are from state register (FSM) to destination register (datapath) through the source operand multiplexer. Here *c1* and *c2* correspond to control logic for the select signals of the respective multiplexers. Path 5 is from state register (FSM) to destination register (datapath) through the control logic *cu* used for selecting the function used in the functional unit. Path 6 is from state register (FSM) to destination register (datapath) through control logic *cd* used to select the destination operand in multiplexer. Note that not all paths are realizable. For example, *mux1* might not exist because there is no need for a multiplexer before the unit's only input. This happens in the case that only one register is ever used as its first operand. In this case, the delay of *mux1* should be taken as 0 in Path 1, and Path 3 should be ignored. As another example, if a unit performs a single function, the controller does not have to generate a control signal for selecting the function to perform; therefore, Path 5 can be ignored.

For every path, time has to be reserved for the correct functioning of the registers. Their sum is called the *sequential overhead*.

$$SeqOverhead = T_{setup} + T_{CQ} + T_{skew}$$

We can, therefore, calculate the worst case delay of all paths in a register transfer as follows:

$$Delay(rt) = SeqOverhead + Max\left[\begin{array}{l} Max\left[\begin{array}{l} Delay(c1) + Delay(mux1), \\ Delay(c2) + Delay(mux2), \\ Delay(cu) \end{array}\right] + Delay(u), \\ Delay(cd) \\ Delay(muxd) \end{array}\right] +$$

Although we can determine the delay of the functional unit by looking up Table 5.1, delays of the multiplexers and the control logic are not readily available. There are two complications. First, as discussed earlier, there are degenerate cases where *mux1, mux2, muxd, c1, c2*, *cd*, and *cu* are not needed. In these cases, they assume a zero delay value. Second, when the multiplexers do exist, their delays depend on

the number of inputs they have. This requires us to find, for each functional unit, the total number of different operands for each input and the total number of different opcodes by scanning the entire RTL representation.

Assuming the delay of all control signals assume a value of T_c, we then have:

$$Delay(cu) = \begin{cases} 0 & |Opcodes(u)| = 1 \\ T_c & |Opcodes(u)| > 1 \end{cases}$$

where

$$Opcodes(u) = \{rt.opcode | \forall rt \in T.[rt.u = u]\}$$

Similarly, we have

$$Delay(c1) = \begin{cases} 0 & |Src1(u)| = 1 \\ T_c & |Src1(u)| > 1 \end{cases}$$

$$Delay(c2) = \begin{cases} 0 & |Src2(u)| = 1 \\ T_c & |Src2(u)| > 1 \end{cases}$$

$$Delay(cd) = \begin{cases} 0 & |Srcd(dest)| = 1 \\ T_c & |Srcd(dest)| > 1 \end{cases}$$

where

$$Src1(u) = \{rt.src1 | \forall rt \in T.[rt.u = u]\}$$
$$Src2(u) = \{rt.src2 | \forall rt \in T.[rt.u = u]\}$$
$$Srcd(dest) = \{rt.u | \forall rt \in T.[rt.dest = dest]\}.$$

Because the numbers of inputs of *mux1, mux2,* and *muxd,* are given by $|Src1(u)|$, $|Src2(u)|$, and $|Srcd(dest)|$ respectively, we can determine their delays by looking up the multiplexer delays, which are listed according to the input count in Table 5.1.

Example 5.16 To estimate the cycle time of the dot product example, we assume $T_c = 5$ FO4.With the method developed previously, we can find the delay of each register transfer and determine that the cycle time of the RTL is 47.4 FO4. In 90nm technology, this is equivalent to 32.4ps*47.4 = 1.54ns. In other words, the maximum speed at which the circuit can run is 649Mhz. Recall that the cycle count is 401, and the work (number of virtual instructions) is 1303, we can conclude that

$$Perf(TinyRTL) = \frac{1303}{401 * 1.54} = 2108MIPS$$

Register transfer	Seq overhead	u	muxd	cd	cu	c1	c2	mux1	mux2	Delay
sts 0, R0	10.0	0.0	2.4	5.0	0.0	0.0	0.0	0.0	0.0	17.4
sts 0, R1	10.0	0.0	2.4	5.0	0.0	0.0	0.0	0.0	0.0	17.4
M.lda R2, R1, A	10.0	10.5	2.4	5.0	0.0	0.0	5.0	0.0	2.4	30.3
M.lda R3, R1, B	10.0	10.5	2.4	5.0	0.0	0.0	5.0	0.0	2.4	30.3
U0.+ R1, R1, 1	10.0	10.0	2.4	5.0	5.0	5.0	5.0	2.4	3.2	33.2
U1.* R2, R2, R3	10.0	35.0	2.4	5.0	0.0	0.0	0.0	0.0	0.0	47.4
U0.< R3, R1, 100	10.0	10.0	2.4	5.0	5.0	5.0	5.0	2.4	3.2	33.2
U0.+ R0, R0, R2	10.0	10.0	2.4	5.0	5.0	5.0	5.0	2.4	3.2	33.2

Of course, other quality metrics such as silicon area and power consumption are also very important. Similar procedures can be developed to estimate these metrics directly from RTL representation and currently are active areas of research.

Walking through the dot product example reveals that to transform an application in behavioral (C/C++) form to RTL (Verilog) form, several key decisions have to be made to convert the application into an IR form and eventually into RTL form. Performance analysis gives us further insight that these decisions impact the performance the RTL implementation can achieve, both in terms of cycle count, and cycle time, in a non-trivial manner. In the next section, we develop CAD algorithms that allow these key decisions to be made to optimize the various quality metrics.

5.3 HIGH-LEVEL SYNTHESIS ALGORITHM OVERVIEW

In the previous section, we outlined the steps required to transform a behavioral description of the dot product algorithm in TinyC into its RTL implementation in Verilog. Although we defined the intermediate program forms, we did not describe how the input form is transformed into the output form. For the frontend and optimizer components, the techniques used are largely no different from those used by software compilers. We refer the readers to [Aho 2006] for a detailed treatment. For the code generator component, it is relatively simple to output the HDL code from the RTL representation. Thus in this section we focus on the backend component containing the core synthesis algorithms that take as input an IR in terms of virtual instructions, and generate as output an RTL representation in terms of register transfers.

As discussed in Section 5.2.5, we are not only interested in generating the correct RTL that is functionally equivalent to the IR, but also with the best ***quality of result*** (QoR). Therefore, the synthesis problem can be formulated as an optimization problem targeting multiple objectives such as performance, area, and power. For example, the backend synthesis problem can be formulated as the following.

PROBLEM 5.1
Given: $TinyIR = \langle O,S,V,B \rangle$
Find: $TinyRTL = \langle M,R,U,I,C \rangle$
Maximize: *Perf(TinyRTL)*
Minimize: *Area(TinyRTL)*

Recall that because this is a complex, phase-coupled optimization, it is not obvious how this problem can be solved. As discussed in Section 5.2, a divide-and-conquer strategy is usually followed, and high-level synthesis is further

decomposed into allocation, scheduling, and binding problems, and solved in sequence. To simplify the presentation, in this section we make further simplifications:

- Assumption 1: The functional unit allocation (*i.e.*, the hardware resources used in the implementation), is specified by the user as a constraint. The rationale behind this assumption is that the user could try out different hardware resource allocations, and let the automated synthesis tool generate solutions for comparison, a process known as *design exploration*. We further assume that each allocated unit can implement all virtual instructions.
- Assumption 2: Storage allocation and binding is performed partially by assuming all array variables are mapped to a single memory, and all scalar variables are mapped to separate registers. Therefore, only the temporary values produced by virtual instructions need to be mapped to registers.
- Assumption 3: Other than the constant instruction, each virtual instruction is implemented by one and only one register transfer. This does not have to be the case in a production synthesis tool, because a subset of virtual instructions can be grouped together and implemented by a single register transfer, a process known as *chaining*.
- Assumption 4: Each register transfer can be completed in a single clock cycle without degrading cycle time. In practice, this is not true because a virtual instruction could be implemented by a functional unit with longer delay than the desired cycle time, a process known as *multicycling*.

We can then refine Problem 5.1 as follows.

PROBLEM 5.2

Given:	TinyIR $= \langle O,S,V,B \rangle$
Find:	(1) Schedule *Sched*: $B \mapsto (V \mapsto Z)$
	(2) Register binding B^R: $V \mapsto Z$
	(3) Functional unit binding B^U: $V \mapsto U$
Minimize:	Objective (1) $\forall b \in B$, $\lvert range\ Sched(b) \rvert$
	Objective (2) $\lvert range\ B^R \rvert$
	Objective (3) $\Sigma_{u \in U} \lvert Src1(u) \rvert + \Sigma_{u \in U} \lvert Src2(u) \rvert + \Sigma_{r \in R} \lvert Srcd(r) \rvert$
Subject to:	Constraint $\forall b \in B, \forall s \in Z, \lvert Sched(b)^{-1}(s) \rvert \leq \lvert U \rvert$

Here we attempt to make three key decisions. For each basic block, the schedule *Sched* maps each instruction contained in the basic block to a control step. The register binding B^R maps the value computed by each instruction to a register. The functional unit binding B^U maps each instruction to a functional unit. The decisions have to satisfy the resource constraint; in other words, the number of instructions scheduled at each control step cannot exceed the number of

available functional units. Combined with the simplifying assumptions, it is trivial to find the TinyIR representation $\langle M,R,U,I,C \rangle$ from *Sched*, B^R, and B^U.

We now relate the objectives in Problem 5.1 to the objectives in Problem 5.2. To maximize performance, Problem 5.2 states that it is equivalent to minimizing the number of control steps in each basic block (objective 1). Recall performance is the product of cycle count and maximum clock frequency. Although cycle count is not the same as control step count (because a control step could be executed many times), minimizing the latter does minimize the former. Here we assumed that maximum clock frequency is independent of scheduling and binding, which does not hold in general. To minimize area, Problem 5.2 states that it is equivalent to minimizing the register count (objective 2), and minimizing the total number of multiplexer inputs (objective 3). This makes sense because the functional units, memory, and registers for scalar variables are pre-allocated and therefore fixed. Here we have assumed that the area of the synthesized circuit is dominated by the datapath (controller area is therefore ignored), and areas of multiplexers are proportional to the input count.

The next two sections describe the scheduling and register allocation steps.

5.4 SCHEDULING

5.4.1 Dependency test

Because the objective of scheduling is to minimize the total number of control steps (*i.e.*, maximize performance), we wish to schedule as many instructions in the same step as possible, thereby executing all of them in parallel to maximize design performance. However, this is not always possible. For an RTL implementation to preserve the semantics of the original algorithm, *data dependencies have* to be respected. We illustrate the notion of data dependency below.

Consider the following scenarios in a basic block, where virtual instruction A precedes virtual instruction B:

1. Instruction A is the operand of instruction B, in other words, instruction A produces a value consumed by instruction B;
2. Instruction A stores a value to symbol x, whereas instruction B loads a value from symbol y;
3. Instruction A loads a value to symbol x, whereas instruction B stores a value to symbol y;
4. Instruction A stores a value to symbol x, whereas instruction B stores a value to symbol y.

For scenario 1, there is a *definite data dependency* between A and B: B has to be scheduled at least one step later than A, because the value of A has to be

produced first before it can be consumed. This relation is explicitly represented in, and easily extracted from, the IR.

For each of the scenarios 2, 3, and 4, there is a *potential data dependency* between A and B: as soon as one can determine that symbols x and y are the same (*i.e.*, they are *aliased* to each other), then B has to be scheduled at least one step later than A. This relation is implicitly induced by the runtime value of memory addresses, and thus not easy to extract from the IR.

A dependency tester is an algorithm that statically determines whether two instructions depend on each other. In TinyRTL, all scalar symbols are explicitly named, in other words, symbols x and y either have the same name, or they are not aliased to each other. It is therefore straightforward to compute the data dependency induced by scalar variables by comparing symbol names. Dependency through indexed accesses to arrays is more difficult to detect. Given array access $x[i]$, and array access $x[j]$, the dependency test amounts to determining whether values i and j can be equal to each other at runtime. The supercomputing research community has developed comprehensive methods for array-based dependency tests [Aho 2006]. A simple dependency tester for TinyRTL can be constructed by simply comparing symbol names, even for array accesses (in other words, conservatively assume that all indices are potentially equal).

In a real-world language (*e.g.*, C/C++), anonymous symbols exist through the use of pointer dereferences. Pointers can be created either by taking the address of a named symbol, or by dynamic memory allocation. Because pointers can be copied, manipulated, and stored as any other value, two pointers in a program can assume the same value at runtime, in which case the corresponding pointer dereferences become aliases. Computing runtime pointer values statically, known as *pointer analysis*, or statically detecting if two pointer dereferences alias, known as *alias analysis*, are both undecidable. Many pointer/alias analysis algorithms have been developed, with varying precision and scalability. The simplest pointer analysis algorithm collects the set of all symbols in the program whose addresses have been taken, as well as the set of all dynamic memory allocation sites, and assumes all pointers in the program can carry one of those values.

To facilitate scheduling, a precedence graph is first constructed to capture the dependency relation among the instructions in a basic block. The precedence graph is named so because it captures the partial order of instructions.

Definition 5.4 *A precedence graph $\langle E,s,t\rangle$ is a polar graph where $E \subseteq V \times V$ is the set of edges, and $s,t \in V$ is the source and sink node, respectively.*

The precedence graph is sometimes also referred to as the *dataflow graph* in the literature. A minor difference is that the dataflow graph captures the data dependency of all instructions in a procedure, whereas the precedence graph is its subgraph for instructions within a basic block. In particular, the source and sink nodes are introduced to lump all instructions defined outside the basic block under consideration. All instructions outside the basic block that are

depended on by the instructions in the basic block are lumped into the source node. All instructions outside the basic block that depend on the instructions in the basic block are lumped into the sink node.

Example 5.17 Consider the TinyC code fragment in Figure 5.6a. The corresponding TinyIR is shown in Figure 5.6b. A simple dependency test algorithm can establish the dependency relation by examining the chain of operations in each instruction. For example, consider instruction (28) in basic block B4, whose chain of operands is shown in Figure 5.7a. It can be inferred that it depends on instruction (26), which in turn depends on instructions (24) and (25), and so on. This process can be repeated, which yields the precedence graph for basic block B4, as graphically depicted in Figure 5.7b. Note that instructions (4) and (6) are defined outside B4, and they are lumped together as source node s. Likewise instructions (27) and (28) are also defined outside B4, and they are lumped together as sink node t. Thus, we have the following edges defined $E = \{\langle s,15\rangle, \langle s,16\rangle, \langle s,24\rangle, \langle s,25\rangle, \langle 15,17\rangle, \langle 16,17\rangle, \langle 24,26\rangle, \langle 25,26\rangle, \langle 17,18\rangle, \langle 18,20\rangle, \langle 20,t\rangle, \langle 26,t\rangle\}$.

```
int a, b, c, d;
                  :
                  :
c = ...;
d = ...;

if( ... ) {
  c = (((a + b) * (a - b)) * 13) + 16;
  d = (a + 12) * (a * 12);
  }

... = c + d;

                  :
                  :
```

(a)

```
scalar c;
scalar d;

B3:
  ...
  (4) lds a
  (6) lds b
  ...
  (13) bt ..., B5
B4:
  (14) cnst 13
  (15) + (4) (6)
  (16) - (4) (6)
  (17) * (15) (16)
  (18) * (14) (17)
  (19) cnst 16
  (20) + (18) (19)
  (23) cnst 12
  (24) + (4) (23)
  (25) * (23) (4)
  (26) * (24) (25)
  (27) sts (20), c
  (28) sts (26), d
B5:
  (30) lds c
  (31) lds d
  (32) + (30) (31)
  ...
```

(b)

FIGURE 5.6

(a) TinyC code. (b) TinyIR representation.

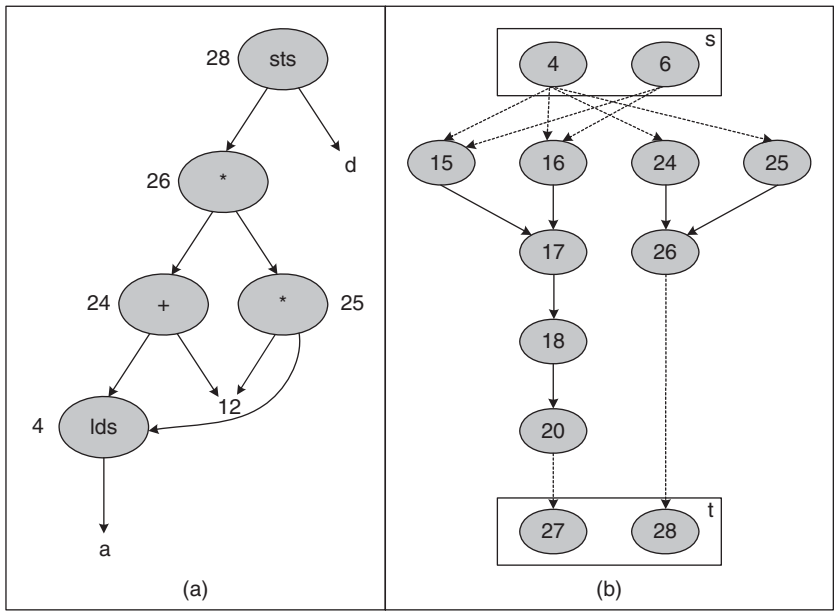


FIGURE 5.7

(a) Chain of instructions. (b) Precedence graph of basic block B4.

We are now ready to examine some commonly used scheduling formulations.

5.4.2 Unconstrained scheduling

We first consider the simple case where the allocation constraints are ignored. In other words, we assume an unlimited number of functional units are available. The unconstrained scheduling problem for a basic block can then be formulated as follows.

PROBLEM 5.3

Given:	*Precedence graph* $\langle E,s,t\rangle$
Find:	$S{:}V \mapsto Z$
Minimize:	$S(t)-S(s)$
Subject to:	$\forall\langle u,v\rangle \in E, S(v) - S(u) > 0$

Assuming the control steps are sequentially numbered, then the total number of steps is defined as $S(t)-S(s)$, which becomes the objective to minimize. To respect data dependency, for every edge $\langle u,v\rangle$, the schedule of the sink, $S(v)$, has to be "later" than the schedule of the source, $S(u)$.

To solve Problem 5.3, one can use an iterative approach. In each iteration, a set of nodes (instructions) in the precedence graph is scheduled to a control step.

The set of nodes that can be scheduled, or are "ready," should be those whose predecessors are all scheduled; otherwise, the dependency constraint would be violated. In addition, we will schedule all nodes as soon as they are ready. This strategy is referred to as ***as-soon-as-possible*** (ASAP) scheduling.

To implement ASAP scheduling, one can maintain a set *Ready*, representing the set of nodes ready to be scheduled in the current control step. Initially, *Ready* contains a single element, s. In each iteration, one chooses a node v from *Ready*, and assigns its schedule as the current control step. In addition, it needs to examine each successor w of v, and add it to *NextReady*, if it becomes ready because v is scheduled. To judge if w becomes ready, one could check if all its predecessors have been scheduled. This results in an algorithm with a complexity of $O(|V|+|E|^2)$.

A better approach is shown in Algorithm 5.1. The key insight is that one only needs to maintain the number of unscheduled predecessors for each node, called *counter* in line 4 of Algorithm 5.1. It is initialized to be the number of incoming edges for each node in line 7-8. When node v is scheduled, this number is decremented for each of its successors w. When this number becomes 0, node w becomes ready (lines 14-16). Algorithm 5.1, therefore, has a complexity of $O(|V|+|E|)$.

Algorithm 5.1 ASAP Scheduling

```
algorithm asapSched(E : (V × V)[ ], s : V, t : V) returns V ↦ Z
 1. var     S : V ↦ Z;
 2. var     Ready, NextReady : V[ ];
 3. var     step : Z;
 4. var     counter : V ↦ Z;
 5. step = −1;
 6. Ready = {s};
 7. foreach (v ∈ V)
 8.   counter(v) = |{u|⟨u,v⟩ ∈ E}|;
 9. while (Ready ≠ ∅) do
10.   NextReady = ∅;
11.   foreach (v ∈ Ready) begin
12.     S(v) = step;
13.     foreach (⟨v,w⟩ ∈ E) begin
14.       counter(w) = counter(w) −1;
15.       If (counter(w) == 0)
16.         NextReady = NextReady ∪ {w};
17.     end foreach
18.   end foreach
19.   step = step + 1;
20.   Ready = NextReady;
21. end while
22. return S;
```

Example 5.18 Consider applying Algorithm 5.1 to the precedence graph shown in Figure 5.7b. The source node s is first scheduled (step -1). Because all its successors {15, 16, 24, 25} have only one predecessor s, they become ready next. Scheduling node 15 to step 0 decrements counter for node 17 from 2 to 1. Scheduling node 16 to step 0 further decrements counter for node 17 from 1 to 0, which triggers node 17 to become ready in the next step. The same would apply to node 24 and node 25 to node 26. So, {17, 26} are scheduled at step 1. In the subsequent iterations, node 18, with node 17 as the only predecessor, is scheduled at step 2; and its dependent, node 20, is scheduled at step 3. The complete ASAP schedule is shown in Figure 5.8a; it shows that it takes 4 steps at minimum to schedule all instructions, excluding s and t.

Note that ASAP scheduling is not the only possible solution to the unconstrained scheduling problem. An equally viable solution is the ***as-late-as-possible*** (ALAP) algorithm. Opposite to the ASAP, the ALAP starts scheduling in reverse time order.

Example 5.19 Consider applying ALAP scheduling to the precedence graph in Figure 5.7b. In the first iteration of the algorithm, all immediate predecessors of the sink will be scheduled at the last step of the schedule. So, {20, 26} are scheduled at step 3. In the subsequent iterations, an instruction is scheduled as soon as all of its successors in the precedence graph have been scheduled at some later step. For example, in the second iteration, node 18 is scheduled at step 2, because it has the already scheduled node 20 as its only successor. Figure 5.8b is the complete ALAP schedule.

5.4.3 Resource-constrained scheduling

We now turn to solving the scheduling problem under resource constraints. In particular, a preallocation of functional units U is available. To simplify discussion,

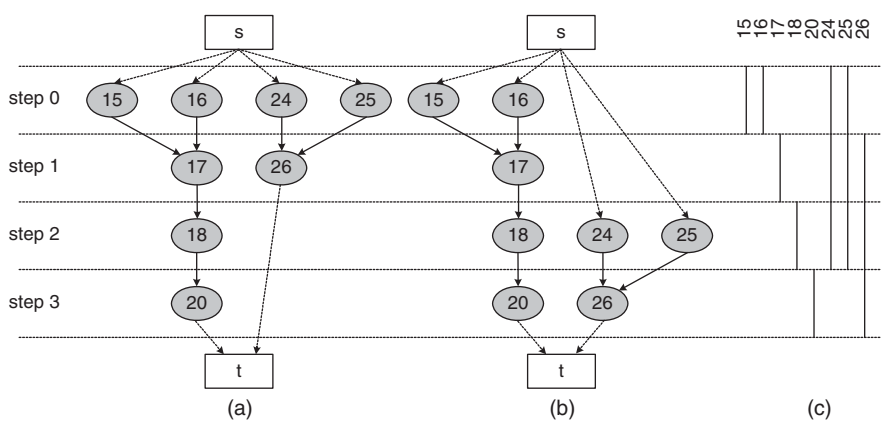


FIGURE 5.8

(a) ASAP schedule. (b) ALAP schedule. (c) Mobility.

we assume each unit $u \in U$ can implement a subset of virtual instructions. The resource-constrained scheduling problem can then be formulated as follows.

PROBLEM 5.4

Given:	Precedence graph $\langle E, s, t \rangle$
Find:	Schedule S: $V \mapsto Z$
Minimize:	$S(t) - S(s)$
Subject to:	Constraint (a) $\forall \langle u,v \rangle \in E : S(v) - S(u) > 0$
	Constraint (b) $\forall i \in Z, \lvert S^{-1}(i) \rvert \leq \lvert U \rvert$

Note that compared with Problem 5.3, a new constraint is added such that the number of instructions scheduled at any control step cannot exceed the number of functional units available. This seemingly simple constraint dramatically changes the combinatorial structure of the problem. Although Problem 5.3 can be optimally solved in linear time, Problem 5.4 is shown to be an NP complete problem and requires heuristics for practical implementations.

Example 5.20 Consider again the scheduling problem that we solved in Example 5.18 and Example 5.19. The precedence graph is redrawn in Figure 5.9, where each node is annotated with its corresponding opcode. Assuming that we only have 2 add/sub units and 1 multiplier, both of our ASAP and ALAP schedules will become infeasible. Referring to the ASAP schedule in Figure 5.8a, our resource constraint is violated by the schedule at step 0 and that at step 1, because 3 add/sub units and 2 multipliers will be needed,

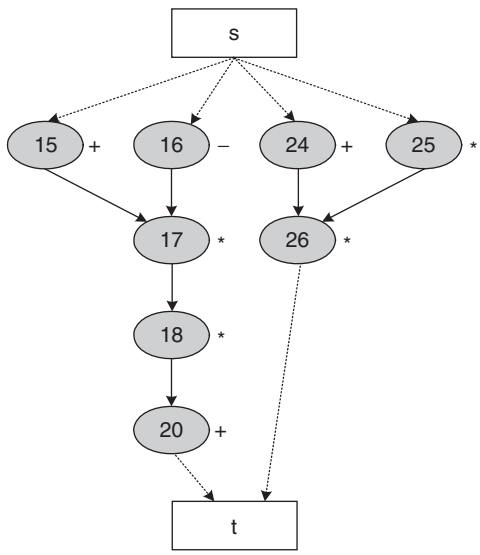


FIGURE 5.9

Precedence graph annotated with opcodes.

respectively. Referring to Figure 5.8b, the schedule at step 2 requires at least 2 multipliers, so it also violates the constraint. To satisfy the resource constraint, we will use a list-scheduling algorithm to schedule these operations again.

List scheduling, shown in Algorithm 5.2, is a modified version of the ASAP scheduling algorithm. Like ASAP, a list of nodes ready for scheduling is maintained, hence its name. The difference is that in each iteration, only a subset of nodes can be scheduled depending on the availability of resources at the current control step. The availability information is maintained with a "reservation table" at line 5 of Algorithm 5.2. We define a Boolean vector *restab*, in which each entry indicates the availability of a unit. At each control step, *restab* is initialized to be all false. Whenever a resource u is available, one of the instructions is selected for assignment to the current step. The case in which a unit can only implement a subset of instructions can be trivially handled by an additional test *impl*. Meanwhile, *restab(u)* is assigned to true, indicating it is "occupied." This ensures constraint (b) is always satisfied.

Algorithm 5.2 List Scheduling

```
algorithm listSched (E : (V × V)[ ], s : V, t : V) returns V ↦ Z
 1. var     S : V ↦ Z;
 2. var     Ready, NextReady : V[ ];
 3. var     step: Z;
 4. var     counter: V ↦ Z;
 5. var     restab: U ↦ {true, false};
 6. step = 0;
 7. Ready = {s};
 8. foreach (v ∈ V)
 9.    counter(v) = |{u|⟨u,v⟩ ∈ E}|;
10. while (Ready ≠ ∅) do
11.  NextReady = ∅;
12.  foreach (u ∈ U)
13.     restab(u) = false;
14.  while (∃u ∈ U,∃y ∈ Ready | !restab(u) ∧ impl(u, y)) do
15.     v = choose(Ready, u);
16.     restab(u) = true;
17.     S(v) = step;
18.     Ready = Ready − {v};
19.     foreach (⟨v,w⟩ ∈ E) begin
20.        counter (w) = counter(w) −1;
21.        If (counter(w) = = 0)
22.           NextReady = NextReady ∪ {w};
23.     end foreach
24.  end while
25.  step = step + 1;
```

26. *Ready* = *Ready* ∪ *NextReady*;
27. **end while**
28. **return** *S*;

Note that at each scheduling step, the number of ready nodes is often more than the resources available to implement them. Therefore, we need to decide which subset of nodes should be chosen for scheduling, in other words, how we implement the *choose* function in line 15 of Algorithm 5.2. It is this key step that impacts the quality of the solution. If a node is chosen too late, then it can potentially lengthen the total schedule. If a node is chosen too early, then potentially more clock steps are needed to keep its value in a register before it is consumed by all its successors. This is referred to as "register pressure," which can lead to an excessive use of registers.

A common way to solve this problem is to assign a priority for each node indicating the desirability of scheduling the node early. The priority can be assigned according to several heuristics or as a weighted sum of them.

One heuristic is to exploit the flexibility of the nodes: in general, there can be potentially many different clock steps a node can be scheduled to. We have already seen that ASAP and ALAP give different solutions, both satisfy the dependency constraint. However, the degree of flexibility can differ. The less-flexible-first heuristics says that we should assign high priority to those nodes that are less flexible. On the other hand, we can afford to schedule highly flexible nodes later because there are more options. To quantify the schedule flexibility, one can use *mobility range*, defined to be the difference between the ALAP schedule and the ASAP schedule for each node.

Example 5.21 The mobility range of all nodes in Figure 5.7b is shown in Figure 5.8c. It can be observed that nodes {15, 16, 17, 18, 20} have zero mobility, whereas the others have a mobility of 2.

We now consider the use of mobility range as the priority function for list scheduling.

Example 5.22 Consider the application of Algorithm 5.2 on the precedence graph in Figure 5.7b, modified in Figure 5.9 with each node annotated with the opcode they require. At step 0, each of {15, 16, 24} in the ready list is requesting an add/sub unit, and {15, 16} are chosen to fill the two available resource slots because both of them have lower mobility than node 24. At the same step, operation 25 is the only candidate requesting a multiplier in the ready list, so it is selected without competition. Such competition happens again at step 2: the multiplier is requested by both ready operations. With lower mobility than operation 26, operation 18

wins. As shown in the following, the schedule generated by this algorithm takes 4 steps, which yields the same minimum latency for ASAP and ALAP scheduling without resource constraints.

Ready List	STEP	ADD/SUB 1	ADD/SUB 2	MULT
{**15, 16,** 24**, 25**}	0	15	16	25
{**24, 17**}	1	24	-	17
{**18**, 26}	2	-	-	18
{**26, 20**}	3	20	-	26

We can also deploy a priority function that selects randomly; it is instructive to compare list scheduling with a mobility-based priority function and with a random priority function.

Example 5.23 Assume the priority function in list scheduling gives random selections. This could lead to two possible schedules. In the first case, nodes {15, 24} are selected to fill the two add/sub slots instead of {15, 16}.

Ready List	STEP	ADD/SUB 1	ADD/SUB 2	MULT
{**15,** 16**, 24, 25**}	0	15	24	25
{**16, 26**}	1	16	-	26
{**17**}	2	-	-	17
{**18**}	3	-	-	18
{**20**}	4	20	-	-

In Figure 5.10b, we can see that the result of delaying operation 16 is taking 1 more cycle than optimal to complete the computation. In the second case, we choose to delay operation 18 at step 3 as shown in the following table, and it also leads to an increase in latency by 1 as shown in Figure 5.10c.

Ready List	STEP	ADD/SUB 1	ADD/SUB 2	MULT
{**15, 16,** 24**, 25**}	0	15	24	25
{**24, 17**}	1	24	-	17
{18, **26**}	2	-	-	26
{**18**}	3	-	-	18
{**20**}	4	20	-	-

Because 16→17→18→20 is a critical path in the precedence graph as shown in Figure 5.9, it is impossible to generate a schedule that takes less than 5 cycles if we delay any node on this path.

Many other heuristics can be developed and are used in practice as well. For example, the distance of a node v to the sink node; in other words, the difference of unconstrained schedules of t and v, wither ASAP or ALAP, can be used as penalty (the inverse of priority). The rationale being that the closer a node is to the sink, the more likely it is to extend the overall schedule if not scheduled early. As another example, the out degree of a node can be used as a priority

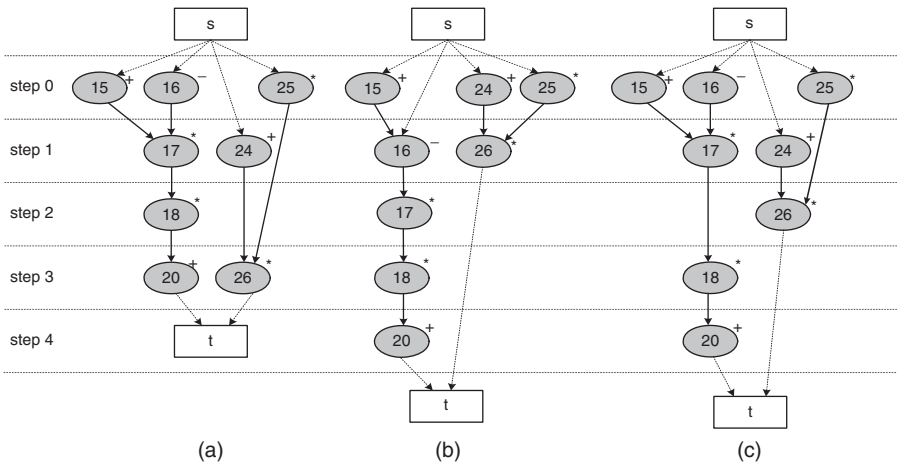


FIGURE 5.10

List scheduling with priority functions.

function: the more number of successors a node has, the more likely additional nodes would become ready after the scheduling of such a node.

5.5 REGISTER BINDING

In an IR, a virtual instruction may compute a *value*, which needs to be kept in a register and later used by other instructions as operands.[2] In a simplistic implementation, one could allocate a distinct register to hold each value. This leads to excessive use of storage resources. In contrast, one could exploit the fact that the values do not need to be held all the time, because they have limited *lifetimes*. The values that have nonoverlapping lifetimes can then share the same register to save silicon area. The task of mapping values to registers to maximize sharing is called *register binding*.

5.5.1 Liveness analysis

To enable register binding, we have to establish the condition under which variables can share a common register.

Definition 5.5 *A value (instruction) v is* ***live*** *at a control step s1 if there exists another control step s2 reachable from s1, such that v is used as an*

[2]Not all virtual instructions compute a value, for example, the memory store instructions. For most instructions that do compute a value, we do not distinguish the virtual instruction and the value it computes.

operand by one of the instructions scheduled at s2. A ***live set*** *at control step s1 is the set of all values alive at s1.*

Clearly, a value v scheduled at control step s cannot share a common register with a value w live at s, otherwise, the new value v would corrupt the value w, that is used later.

We use a liveness analysis algorithm to compute the live set. We first consider a single basic block.

The strategy we take is to start at the end of the schedule and scan each control step backwards (in reverse order). At each control step s, we define,

Live(s):	Set of live values at the beginning of step s
Def(s):	Set of values defined at step s
Use(s):	Set of values used at step s.

The relationship between them can be established as follows, assuming all control steps are sequentially numbered.

$$Live(s) = Use(s) \cup [Live(s+1) - Def(s)]$$

Note that, *Def(s)* is the set of all instruction scheduled at step s; and *Use(s)* is the set of all operands used by instructions scheduled at step s.

A liveness analysis algorithm for a basic block can then be developed, as shown in Algorithm 5.3. It takes the schedule of the basic block as input. An additional input to the algorithm is the set of live values leaving the basic block, called *liveOut*.

Algorithm 5.3 Basic Block Liveness Analysis

```
algorithm liveBB (S : V ↦ Z, liveOut : V[ ]) returns Z ↦ V[ ]
1. var    Live: Z ↦ V[ ];
2. var    l : Z;
3. l = | range S |; Live(l) = liveOut;
4. foreach (s ∈ [l−1...0]) begin
5.    Live(s) = Live(s + 1)−S^−1(s);
6.    foreach (v ∈ S^−1(s))
7.       Live(s) = Live(s) ⋃ {v.src1} ⋃ {v.src2};
8. end foreach
9. return Live;
```

Example 5.24 Consider the scheduled basic block in Example 5.22, re-created in Figure 5.11a. The live range of each value can be visualized by an interval, starting from just after the control step when it is defined and ending at the control step when it is last used as shown in Figure 5.11b. The live set of each control step can then be visualized as a horizontal cut line through the live ranges, in other words, the set of all values crossing the control

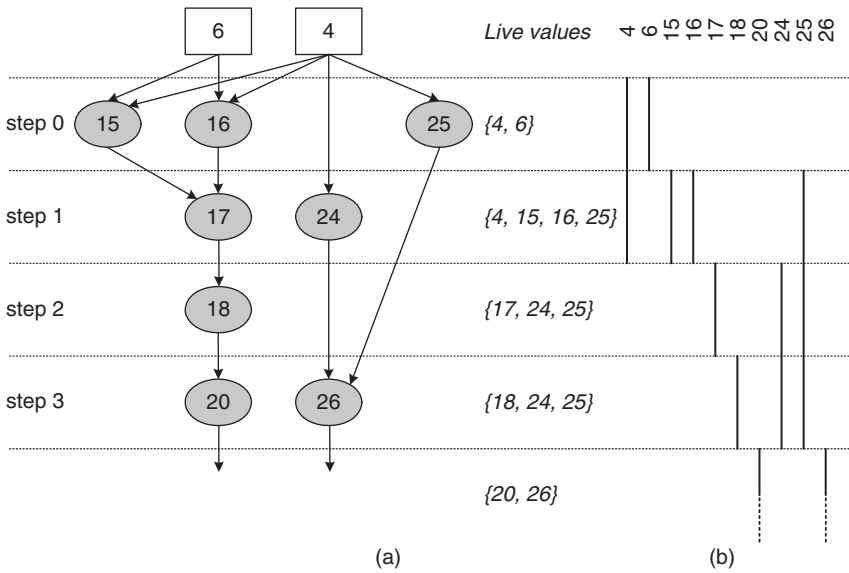


FIGURE 5.11

(a) Results of liveness analysis. (b) Live ranges of values.

step. We now consider how Algorithm 5.3 can compute the correct live set. In the beginning, the liveOut value is {20, 26}, because they are used by other basic blocks. As we scan step 3, we remove {20, 26} from the live set (Line 5, where $S^{-1}(s)$ applies the inverse function of schedule S on step s, or returning the set of a value scheduled at step s), because they are defined in step 3, and add {18, 24, 25}, because they are used by {20, 26}. This leaves {18, 24, 25} as the live set for step 3. This process repeats until we reach step 0, where the live set is {4, 6}, defined in other basic blocks.

STEP	*Def*	*Use*	*Live*
0	{15, 16, 25}	{4, 6}	{4, 6}
1	{17, 24}	{4, 15, 16}	{4, 15, 16, 25}
2	{18}	{17}	{17, 24, 25}
3	{20, 26}	{18, 24, 25}	{18, 24, 25}
4			{20, 26}

We now extend liveness analysis to the whole program. As shown in Algorithm 5.4, we are now given the schedule for all basic blocks and attempt to find the live set for each control step in all basic blocks. We use a standard dataflow analysis framework deployed for software compiler analysis. In this framework, a ***control flow graph*** (CFG) is constructed so that there is one edge from basic block A to basic block B if there is an instruction in block A that branches or jumps to B. The framework traverses all basic blocks following

the CFG order and derives the information of interest by processing each basic block. In this case, we use the post depth first order to make sure all successors of a basic block are processed before a given basic block. As each basic block is processed by calling *liveBB* (in Line 12), the *liveOut* of the basic block is computed by combining the set of live values flowing out of all its successors (Lines 8–13). This process repeats and is terminated when reaching a fixed point; in other words, the computed live set values no longer change.

Algorithm 5.4 Liveness Analysis

```
algorithm live(Sched: B ↦ (V ↦ Z)) returns B ↦ (Z ↦ V[ ])
 1.  var     Live: B ↦ (Z ↦ V[ ]);
 2.  var     LiveOut: B ↦ V[ ];
 3.  var     New: V[ ];
 4.  var     changed: {true, false} = true;
 5.  while (changed) do
 6.    changed = false;
 7.    foreach (b ∈ B in postorder) begin
 8.      New = ⋃_{s∈succ(b)} Live(s,0);
 9.      if (New ≠ LiveOut(b)) begin
10.        LiveOut(b) = New;
11.        changed = true;
12.        Live(b) = liveBB(Sched(b), liveOut(b));
13.      end if
14.    end foreach
15.  end while
16.  return Live;
```

With liveness information, we can then capture the relation between values with a graph, called an *interference graph*. In an interference graph, a node represents a value, and an edge between two nodes indicates that they cannot share a common register. The interference graph can be derived from liveness information with Algorithm 5.5.

Algorithm 5.5 Interference Graph Construction

```
algorithm intf(Sched: B ↦ (V ↦ Z), Live: B ↦ (Z ↦ V[ ])) returns (V × V)[ ]
1. var      E_intf : (V × V)[ ];
2. foreach (b ∈ B) foreach (v ∈ b) begin
3.    s = Sched(b, v);
4.    E_intf = E_intf ∪ {⟨u,v⟩,⟨v,u⟩|u ∈ Live(b,s + 1) ∧ u ≠ v};
5. end foreach
6. return E_intf;
```

Example 5.25 Figure 5.12 shows the interference graph constructed from the liveness information in Example 5.24. For example, for instruction 20, it is scheduled at control step 3. The live set for step 4 is {20, 26}. Therefore an interference graph edge between 20 and 26 is created. This process is repeated for every instruction.

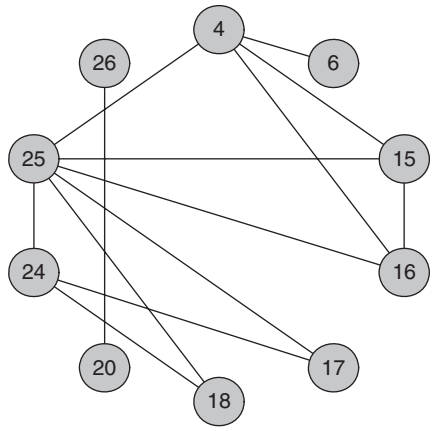


FIGURE 5.12

Interference graph.

5.5.2 **Register binding by coloring**

Given an interference graph, the register-binding problem reduces to assigning each node to a register number, while ensuring that two nodes connected by an edge are assigned different register numbers. This is equivalent to the classic *graph coloring* problem, if each register number is treated as a color.

PROBLEM 5.5

Given:	Interference graph E_{intf}
Find:	Register binding $B^R: V \mapsto Z$
Minimize:	$\lvert$ range $B^R \rvert$
Subject to:	$\forall \langle u,v \rangle \in E_{intf}, B^R(u) \neq B^R(v)$

Minimizing the number of registers is then equivalent to minimizing the *chromatic number*, or the minimum number of colors used to color the interference graph.

The coloring problem is an NP-complete problem and thus requires heuristic solutions. A typical heuristic algorithm colors one node at a time by choosing the minimum color not used by its neighbors. Of course, one can only choose a color different from the neighbors that have already been colored. Therefore,

it is sufficient to consider only the *remainder graph* (*i.e.*, the subgraph of the interference graph where all uncolored nodes and their incident edges are removed).

Algorithm 5.6 uses this strategy by first finding the so-called *vertex elimination order* σ (Line 6), which can be considered as the order in which the sequence of the remainder graph is generated by removing one node at a time starting from the full interference graph. The inverse of the vertex elimination order is, therefore, the order in which the nodes are colored. In fact, in each iteration of the loop in Line 7, a node v is selected according to σ and added to the remainder graph (Lines 9–10) and colored (Line 11).

Algorithm 5.6 Register Binding by Coloring

algorithm *color*(E_{intf} : ($V \times V$)[ ]) **returns** $V \mapsto \mathbf{Z}$
1. **var** $C : V \mapsto \mathbf{Z}$;
2. **var** $\sigma : V \mapsto \mathbf{Z}$;
3. **var** $V' : V[\,]$;
4. **var** $E' : E_{intf}[\,]$;
5. **var** $v : V$;
6. $\sigma = vertexElim(E_{intf})$;
7. **foreach** ($i \in [1 \ldots |V|]$) **begin**
8. $v = \sigma^{-1}(i)$;
9. $V' = V' \cup \{v\}$;
10. $E' = E' \cup \{\langle u,v\rangle, \langle v,u\rangle \mid u \in V' \wedge \langle u,v\rangle \in E_{intf}\}$;
11. $C(v) = \min(\{c \in \mathbf{Z} \mid \forall \langle u,v\rangle \in E', C(u) \neq c\})$;
12. **end foreach**
13. **return** C;

Example 5.26 Consider the interference graph in Example 5.25. Assume the vertex elimination order is {6, 20, 26, 17, 18, 24, 4, 15, 16, 25}. Then the coloring order is {25, 16, 15, 4, 24, 18, 17, 26, 20, 6}. We start with an empty remainder graph. The first node chosen is node 25. Because it has no neighbors, color 1 is chosen. In the next iteration, node 16 is chosen, and the remainder graph is expanded with the node, as well as the incident edges, shown in Figure 5.13a. Color 2 is chosen as the minimum number besides 1. In the next iteration, node 15 is added to the remainder graph, shown in Figure 5.13b, and assigned color 3, the minimum color different from its neighbors 25 and 16. This process is repeated in Figure 5.13b–e. In the end, all 10 nodes in the interference graph are colored with 4 colors. In other words, 10 values can share 4 registers. The mapping is as follows.

R0: {6, 25, 26}
R1: {16, 20, 24}
R2: {15, 17, 18}
R3: {4}

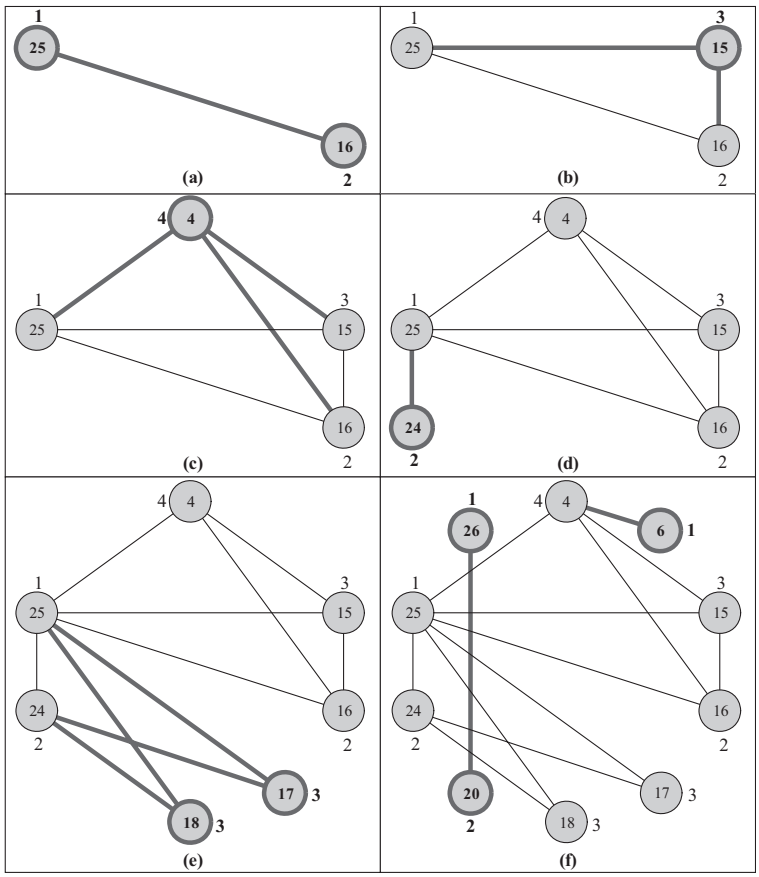


FIGURE 5.13

Coloring with $\sigma = \{6, 20, 26, 17, 18, 24, 4, 15, 16, 25\}$.

Although the coloring process itself is rather straightforward, the key step requires computing the appropriate vertex elimination order, or the coloring order.

We first consider the special case in which the interference graph is derived from a basic block. As shown in Example 5.24, each value is associated with a live range characterized by a control step when the value is defined, and a control step when it is last used. This live range can be considered as an interval on the integer set. An interference graph constructed by creating one edge for each pair of overlapping intervals is called an *interval graph*. For an interval graph, one could pick a coloring order by sorting all intervals according to the left edge of the interval. This *left-edge algorithm* is optimal for an interval graph.

We now consider the general case in which the interference graph may not have the structural property of an interval graph. This may partly be due to the presence of complex control structures, such as branches and loops. This may also be due to the special requirements that certain values have to be mapped to the same register.

A typical heuristic we can use is called less-flexible-first, which we have used in list scheduling. Here, the more neighbors a node has, the less choices it may have to assign a color, and, therefore, the earlier we should color it. We can, therefore, pick a vertex elimination order according to the degree of a node. This strategy is used in Algorithm 5.7.

Algorithm 5.7 Vertex Elimination

algorithm *vertexElim* ($E_{\text{intf}} : (V \times V)[\,]$) **returns** $V \mapsto \mathbf{Z}$
1. **var** $\sigma : V \mapsto \mathbf{Z}$;
2. **var** $V' : V[\,]$;
3. **var** $E' : (V \times V)[\,]$;
4. **var** $v : V$;
5. $V' = V$;
6. $E' = E_{intf}$;
7. **foreach** ($i \in [1..|V|]$) **begin**
8. $v = \text{argmin}_{v \in V'} \mid \{\langle u,v\rangle \in E'\}|$;
9. $\sigma(v) = i$;
10. $V' = V' - \{v\}$;
11. $E' = E' - \{\langle u,v\rangle, \langle v,u\rangle \in E'\}$
12. **end foreach**
13. **return** σ;

Example 5.27 The vertex elimination process is illustrated in Figure 5.14. Here we use a stack to keep track of the vertex elimination order, which facilitates the use of its inverse as coloring order. With one adjacent node, 6, 20, and 26 are pushed to the stack first, as shown in Figure 5.14a. These nodes, as well as their incident edges, are removed, which results in a remainder graph shown in Figure 5.14b. With the degree of 2, vertices 17 and 18 score highest to be the next two nodes to be eliminated. After removing both 17 and 18, vertex 24 has its degree reduced to 1 and becomes the next candidate to be removed, as shown in Figure 5.14c. As we can see in Figure 5.14d, the remaining nodes have the same degree, and vertex 4 is selected arbitrarily. The same happens in the subsequent iterations, and we choose to push vertices 15, 16, and 25 to stack in order, as shown in Figure 5.14e–f. With the vertex elimination order stored in the stack, coloring can be performed by popping one node at a time from the stack and reconstructing the remainder graph, as illustrated in Example 5.25.

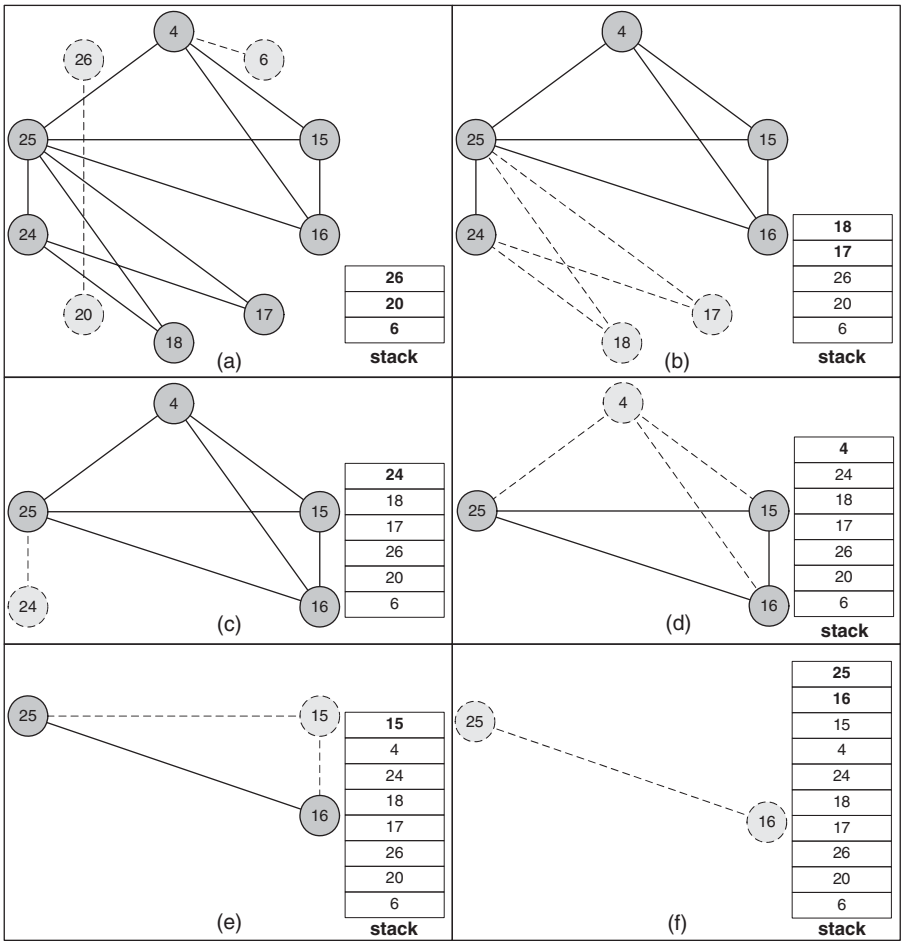


FIGURE 5.14

Vertex elimination.

5.6 FUNCTIONAL UNIT BINDING

Although the coloring algorithm can minimize the register count, the register sharing decision may affect the size of multiplexers. The silicon area of multiplexers can be quite significant. For example, in an ***application-specific integrated circuit*** (ASIC), the area of a six-input multiplexer is close to the area of an adder. For today's FPGAs, the area of multiplexers, typically implemented as lookup tables, is often larger than adders, typically available as hardwired logic.

Example 5.28 To see how register binding may affect multiplexer area, consider the code fragment in Figure 5.15a. Assume both instructions will be mapped to the same adder. If all possible input operands, which are values {*t0, t1, t3, t4*}, are mapped to different registers, as shown in Figure 5.15b, two 2-to-1 multiplexers will be needed. However, as shown in Figure 5.15c, the multiplexers can be eliminated by mapping values {*t0*, *t3*} to the same register and values {*t1, t4*} to another one. In general, even in the cases that only a subset of possible input operands share the same register, there is still a benefit from a smaller multiplexer. Sharing registers among the possible output values of a functional unit can also simplify the interconnection between the functional unit and the registers. This is as illustrated by the mapping of values {*t2, t5*} to register R3 in Figure 5.15c.

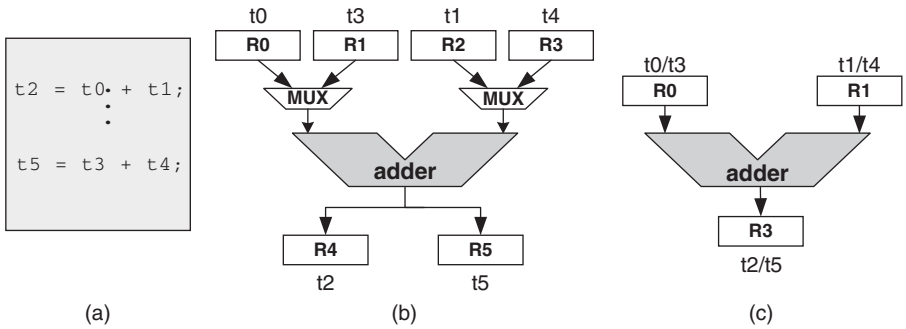


FIGURE 5.15

Impact of register sharing on multiplexers.

Likewise, although the silicon area of functional units is determined by allocation, functional unit binding affects the area of multiplexers. Ideally, if the source operand of one instruction A maps to the same register of the source operand of another instruction B, then it is desirable to map A and B to the same functional unit, because no extra multiplexer input needs to be introduced at the functional unit input. This scenario is called *common source*. Likewise, if the destination operand of instruction A maps to the same register as the destination operand of another instruction B, then it is desirable to map A and B to the same functional unit, because no extra multiplexer input needs to be introduced at the destination register input. This scenario is called *common destination*.

Example 5.29 Consider the design in TinyRTL in Figure 5.16a. Here two addition operations scheduled at C0 are bound to U0 and U1, respectively; whereas the one scheduled at C1 is bound to U0. To share unit U0, as shown in Figure 5.16b, three 2-input multiplexers are

needed in the corresponding datapath. However, because the operation scheduled at C1 shares common sources and common destinations with the other scheduled operation at C0, the multiplexers can be eliminated by binding both instructions to the same unit, as shown in Figure 5.16c.

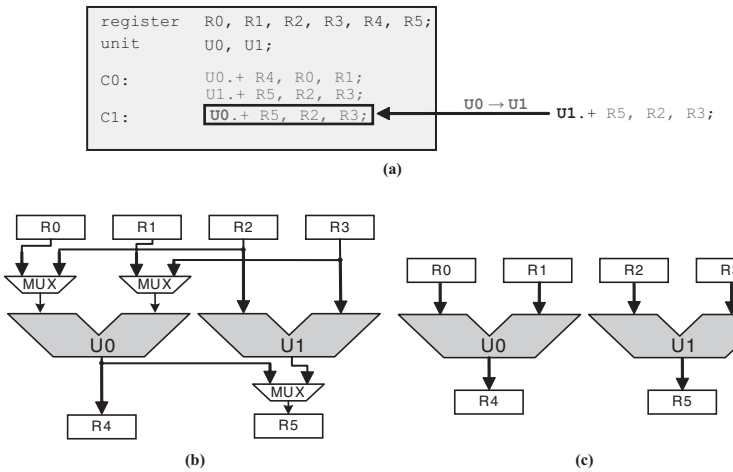


FIGURE 5.16

Impact of functional unit binding on multiplexers.

Mathematically, the register binding and functional unit binding problems for multiplexer area minimization are equivalent, although they depend on each other (called the phase ordering problem). Recall in Section 5.5 that the register-binding problem can be formulated as the coloring of an interference graph. Each edge indicates that the connected nodes cannot be assigned the same color, or they cannot share the same register. We now take a different perspective. Two instructions are said to be *compatible* if they can be mapped to the same resource. Like the interference graph, we can establish a *compatibility graph* whose nodes are instructions, and edges between nodes indicate that they are compatible. The binding problem can then be formulated as a clique partitioning problem, or an integer labeling of nodes in the compatibility graph, such that all nodes with the same label are fully connected to each other. In other words, all nodes with the same label form a *clique*, or a complete subgraph of the compatibility graph.

Interestingly, the interference graph and compatibility graph are dual of each other: one can find the compatibility graph as the inverse of interference graph and vice versa, and clique partitioning of a compatibility graph is equivalent to the coloring of the corresponding interference graph. Thus, an alternate way of performing register binding is to inverse the interference graph to obtain the

compatibility graph, and then solve the clique partitioning problem. Likewise, we can solve the functional unit binding problem by clique partition. The compatibility graph for functional unit binding can be established directly: an edge is created for every pair of instructions that satisfy the following:

- Their opcodes are of the same class.[3]
- They are not scheduled at the same control step.

Example 5.30 Consider the example in Figure 5.6. Consider only the instructions that perform additions and subtractions, which belong to the same class. This gives the set of instructions {15, 16, 20, 24}. Given the schedule in Example 5.22, shown in Figure 5.10a, we conclude that only 15 and 16 interfere with each other, because they are scheduled at the same clock step and, as a result, cannot be bound to the same functional unit. This is reflected in the interference graph in Figure 5.17a. The compatibility graph is shown in Figure 5.17b, essentially by complementing the edges in the interference graph.

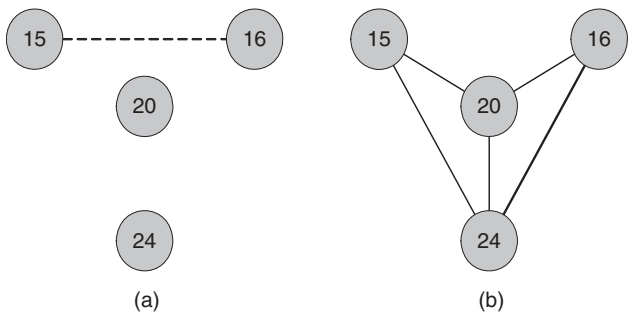


FIGURE 5.17

Interference graph and compatibility graph.

We can then refine functional unit binding problem as follows.

PROBLEM 5.6

Given:	Compatibility graph E_{comp}
Find:	Unit binding B^U: $V \mapsto U$
Minimize:	$\Sigma_{u \in U}\lvert Src1(u)\rvert + \Sigma_{u \in U}\lvert Src2(u)\rvert + \Sigma_{u \in U}\lvert Dest(u)\rvert$
Subject to:	$\forall u,v \in V, B^U(u) = B^U(v) \Rightarrow \langle u,v \rangle \in E_{comp}$

[3]Two instruction opcodes are of the same class if there is significant chance of logic sharing if they are bound to the same functional unit. Addition and subtraction are of the same class, because it requires only a few extract logic gates to convert an adder into an adder/subtractor.

Note that the third term $\Sigma_{u \in U}|Dest(u)|$ of the objective in Problem 5.6, where $Dest(u)$ is defined as the different registers a functional unit u connects to, is derived from $Srcd(r)$ of each register r from the third term of objective (3) in Problem 5.2, where $Srcd(r)$ is the different functional units outputs to register r. In fact, they are just different ways of estimating the total number of multiplexer inputs to the registers.

Like graph coloring, clique partitioning that minimizies the number of cliques is also an NP-hard problem. Problem 5.6 is harder, because its objective is the total number of multiplexers, which nontrivially depends both on functional unit binding and register binding.

To solve the problem heuristically, we again take the iterative approach, which makes one decision at a time. More specifically, as we show in Algorithm 5.8, we start by assuming each node in the compatibility graph forms its own clique. In each iteration, we select and *contract* one edge $\langle u,v \rangle$ in the graph (Lines 11–17); in other words, we merge the pair of nodes incident to the edge into a larger clique. With some bookkeeping (Line 16), the larger clique is represented by one of the pair, say u; therefore, edges incident to the other node v are removed (Line 14). To ensure further merging leads to cliques, any edges incident to u that do not share a common neighbor with v should also be removed (Line 13). This process repeats until all edges are removed, and what is left is a set of nodes, each representing a clique. After this, all virtual instructions in a clique v are assigned a common unit u (line 18–22).

The key step of the algorithm is the criterion used to select the edge in each iteration, so that it positively improves, if doesn't optimizes, our objective. Algorithm 5.8 uses the *partial binding result* to approximate the objective. It assumes that each clique corresponds to a functional unit and maintains its *Src1*, *Src2*, and *Dest*, calculated as the set of registers for the corresponding operands of all nodes in the clique they are mapped to, according to register binding B^R. These sets are called the *operand sets*. In each iteration, when two nodes (cliques) are merged, their corresponding operand sets are merged as well by unions (Line 15). With operand sets defined for each node, we can, in turn, define the *edge weight* as the total number of common operands in respective operand sets. The edge that leads to the least changes, that is, having the most number of common sources and destinations, is greedily selected (Line 11).

Algorithm 5.8 Weighted Clique Partitioning

---

algorithm *CliquePartition*(E_{cmpat} : $(V \times V)[\,]$, B^R : $V \mapsto \mathbf{Z}$) **returns** $V \mapsto \mathbf{Z}$

1. **var** V' : $V[\,] = V$;
2. **var** E' : $(V \times V)[\,] = E$;
3. **var** *Clique* : $V \mapsto V[\,]$;
4. **var** B^U : $V \mapsto \mathbf{Z}$;
5. **var** *Src1*,*Src2*,*Dest* : $V \mapsto \mathbf{Z}[\,]$;
6. **foreach** ($v \in V'$) **begin**

```
7.   Src1(v) = {B^R(v.src1)}; Src2(v) = {B^R(v.src2)}; Dest(v) = {B^R(v.dest)};
8.   Clique(v) = {v};
9.  end foreach
10.  while (E' ≠ ∅) do
11.   ⟨u,v⟩ = argmax_⟨u,v⟩ ∈ E' | Src1(u) ∩ Src1(v)| + |Src2(u) ∩ Src2(v)|
                                + |Dest(u) ∩ Dest(v)|;
12.   V'=V'−{v};
13.   E' = E'−{⟨u,w⟩,⟨w,u⟩|⟨w,v⟩ ∉ E'};
14.   E' = E'−{⟨v,w⟩,⟨w,v⟩ ∈ E'};
15.   Src1(u) = Src1(u) ∪ Src1(v); Src2(u) = Src2(u) ∪ Src2(v);
      Dest(u) = Dest(u) ∪ Dest(v);
16.   Clique(u) = Clique(u) ∪ Clique(v);
17.  end while
18.  foreach (v ∈ V') begin
19.   u = next(U);
20.   foreach (w ∈ Clique(v))
21.     B^U(w) = u;
22.  end foreach
23.  return B^U;
```

We now illustrate the application of Algorithm 5.8 for Example 5.30. We start by computing the initial operand sets.

Example 5.31 From the TinyIR representation in Figure 5.6b, we can find the source operands of the instructions as follows,

INSTRUCTION	SRC 1	SRC 2
15	4	6
16	4	6
20	18	‹16›
24	4	‹12›

Note: constant inputs are bracketed by ‹ ›

From the register allocation result of the previous section (Example 5.26), we have

R0: {6, 25, 26}
R1: {16, 20, 24}
R2: {15, 17, 18}
R3: {4}

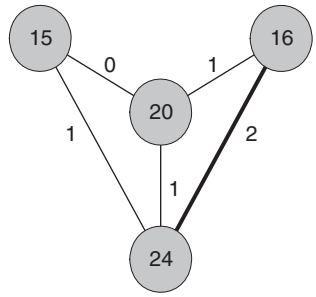
We can, therefore, establish the operand sets in Figure 5.18a. In addition, we mark each edge with a weight, valued as the number of common elements in the respective operand sets.

We can now start the iterative clique partitioning process.

Iteration 1

INSTRUCTION	SRC 1	SRC 2	DEST
15	R3	R0	R2
16	R3	R0	R1
20	R2		R1
24	R3		R1

(a)

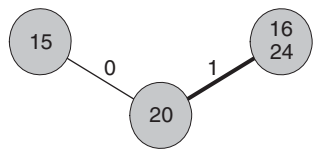


(b)

Iteration 2

INSTRUCTION	SRC 1	SRC 2	DEST
15	R3	R0	R2
16, 24	R3	R0	R1
20	R2		R1

(c)

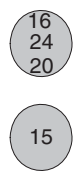


(d)

After iteration 2

INSTRUCTION	SRC 1	SRC 2	DEST
15	R3	R0	R2
16, 24, 20	R3, R2	R0	R1

(e)



(f)

FIGURE 5.18

Operand sets and iterative clique partitioning.

Example 5.32 From Figure 5.18a, the edge ⟨*16,24*⟩ is the only edge with the maximum weight of 2. So in the first iteration it is selected first for contraction. In doing so, nodes 16 and 24 are merged into one node. Note in particular that the original edge ⟨*15,24*⟩ is removed, because 15 and 16 are not compatible. Also note that the operand sets are updated in Figure 5.18c, as well as the edge weights in Figure 5.18d. In the second iteration, the edge between 20 and the merged node in the first iteration has the maximum weight of 1 and is selected next for contraction. The result of this step is shown in Figure 5.18e-f. Examining the remainder of the compatibility graph, there are no more edges left, and the iterative process terminates.

We now have enough information to generate the full datapath.

Example 5.33 Combining scheduling, register binding, and functional unit binding, we can summarize the decisions made in high-level synthesis by the following tables.

REGISTER	VALUES
R0	6, 25, 26
R1	16, 20, 24
R2	15, 17, 18
R3	4

UNIT	INSTRUCTIONS
ADD/SUB 0	16, 20, 24
ADD/SUB 1	15
MULT	17, 18, 25, 26

With the resource binding result, we can complete the datapath of the design by adding multiplexers before functional unit and register input ports. To accomplish this, we need to identify the set of all possible unit-to-register and register-to-unit transfers, given the virtual instructions, as well as the binding result. According to the resource binding result in the above tables, we can identify the sources of registers as in the following table. For example, register R0 takes values {25, 26} from the MULT unit and takes value 6 from an external input, so it needs a 2-input multiplexer to take values from the both sources.

REGISTER	INPUTS	VALUES
R0	external input	6
	MULT	25, 26
R1	ADD/SUB 0	16, 20, 24
R2	ADD/SUB 1	15
	MULT	17, 18
R3	external input	4

To identify the sources of functional units, we first identify the source registers of each instruction, as follows,

INSTRUCTION	SRC 1	SRC_{REG} 1	SRC 2	SRC_{REG} 2
15	4	R3	6	R0
16	4	R3	6	R0
17	15	R2	16	R1
18	‹13›		17	R2
20	18	R2	‹16›	
24	4	R3	‹12›	
25	‹12›		4	R3
26	24	R1	25	R0

Note: constant inputs are bracketed by ‹ ›

Then, the sources of each functional unit are the union of the sources of all instructions bound to the unit. For example, the sources of input port 1 of the add/sub 0 unit is the union of {R3}, {R2}, and {R3}, which are the corresponding sources of instructions 16,

20, and 24, respectively. So, a 2-input multiplexer is needed at the port. The sources of the rest of the functional unit ports are summarized as follows,

UNIT	INPUT PORT 1	INPUT PORT 2
ADD/SUB 0	R2, R3	R0, ‹12›, ‹16›
ADD/SUB 1	R3	R0
MULT	R1, R2, ‹12›, ‹13›	R0, R1, R2, R3

The complete synthesized datapath is as illustrated in Figure 5.20a.

It is constructive to examine whether the heuristic in Algorithm 5.8 is effective. To see this, we apply the same clique partitioning process, except that in each iteration, an edge is randomly selected for contraction. Figure 5.19 shows a possible result, and the corresponding datapath is as illustrated in Figure 5.20b. It shows that the random unit binding–based datapath takes 20 multiplexer inputs compared with 17 multiplexer inputs from the clique-partitioning unit binding. In terms of wiring complexity at the output port, it has a total of 16 destinations from all sources, whereas the clique partitioning unit binding synthesizes a datapath with only 15, which shows that in this instance the clique partitioning heuristic yielded a superior design.

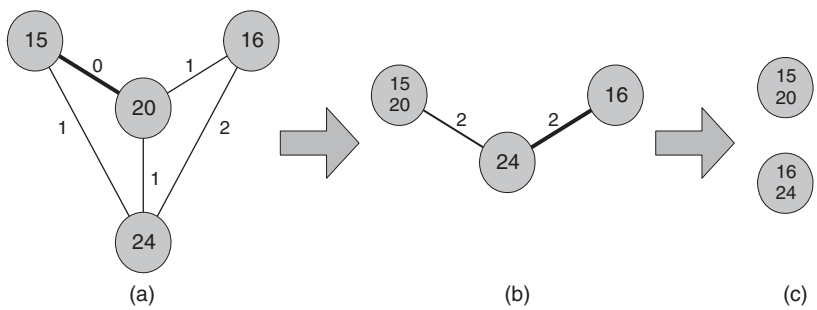


FIGURE 5.19

Random unit binding.

5.7 CONCLUDING REMARKS

In this chapter, we described a complete, although simplified high-level synthesis system. The presented algorithms are distilled from a rich body of research since the 1980s. The representative early academic efforts include CMU [McFarland 1978; Gyrczyc 1984; Thomas 1988], IMEC [De Man 1986], USC [Parker 1986], and Illinois [Pangrle 1987; Brewer 1988]. The representative early industry efforts include IBM [Camposano 1991] and Bell Lab [Bhasker 1990]. Readers are referred

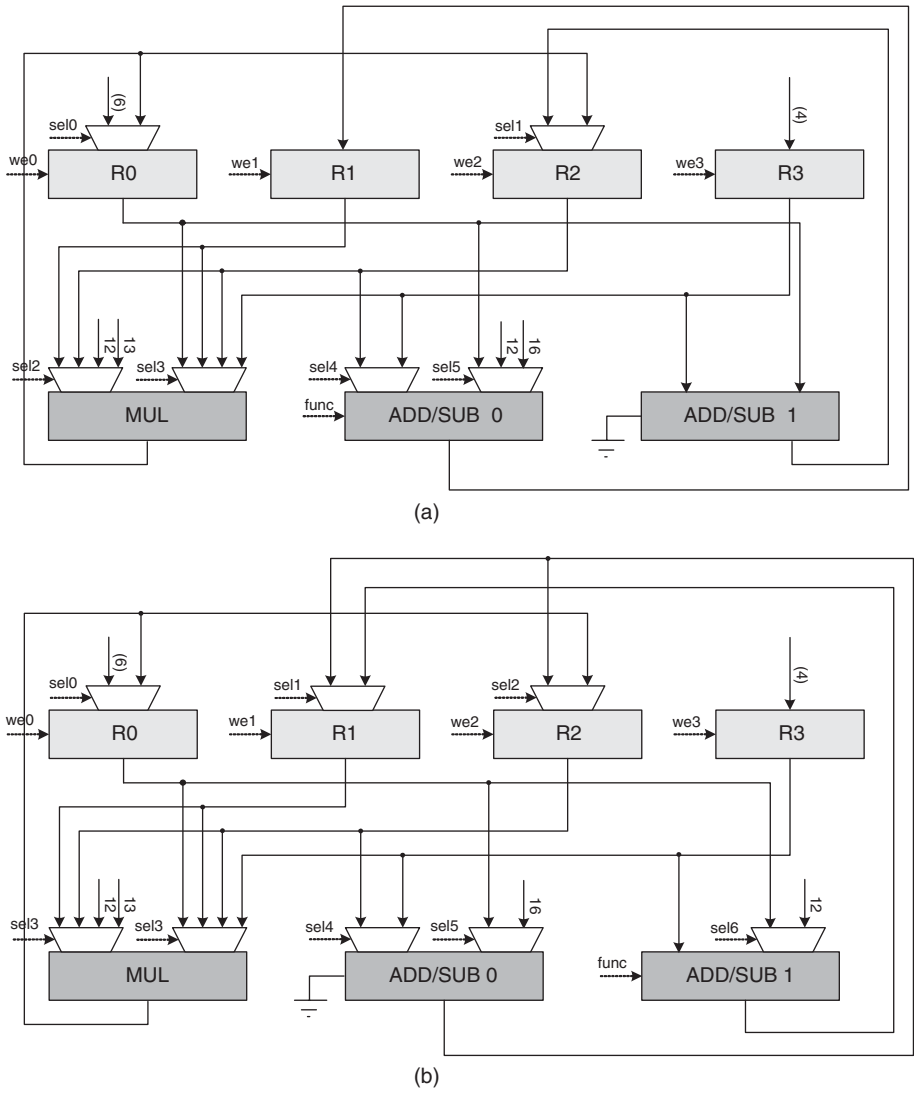


FIGURE 5.20

Synthesized datapaths.

to [Gajski 1992] and [De Micheli 1994] for a comprehensive treatment of development in this period. In particular, list scheduling is due to [Landskov 1980]. Left edge algorithm for register binding is due to [Kurdahi 1987]. Clique partitioning based binding is due to [Tseng 1986].

Although in this chapter we chose to present a resource-constrained–based formulation of high-level synthesis, a large body of literature was devoted to

performance-constrained (or time-constrained) formulations of scheduling. A representative work in this area is force-directed scheduling [Paulin 1989], which attempts to minimize resource count under a cycle count constraint of a basic block.

Despite the intensive research efforts, high-level synthesis was not embraced by the design community as much as it was intended. As discussed in Section 5.1, one reason behind the reluctance of acceptance is methodologic: because the chip content has become increasingly heterogeneous, high-level synthesis has to be integrated into an electronic system–level design method before it can replace register transfer level synthesis as the dominant synthesis technology. Toward this end, *hardware/software codesign* [Gupta 1992; Gajski 1994] emerged in the mid-1990s as a research field to address the issue of how one can partition an application into software and hardware components, select the processors, and generate the interface between them. Attempts to fully automate the task, called *hardware/software cosynthesis* [Ernst 1996; Yen 1996], largely failed in practice because the partitioning decisions are often dictated by non-technical factors, such as the availability of legacy ***intellectual property*** (IP) components that populate the ecosystem of processor IPs. Although the cosynthesis approach will remain effective for domain-specific subsystems, the full system-level design method has given way to the architecture-based method described in Section 5.2, which gives more discretion to system architects, enables concurrent development of hardware and software, and allows for derivative implementations to amortize the design cost to multiple products.

A restricted form of hardware/software codesign that has gained success in practice is ***application-specific instruction set processor*** (ASIP) design. Here a programmable processor is designed; however, the instruction set is adapted to a family of applications. The main attractiveness of ASIPs is that it allows for post-silicon programmability (which is not possible for traditional high-level synthesis that generates custom hardware); furthermore, ASIPs can achieve a certain level of application acceleration through the use of custom instructions. The ASIP based design methodologies pioneered in [Marwedel 1986], [Fauth 1995], [Halambi 1999], and [Hoffmann 2001] are summarized in [Mishra 2008], where different forms of ***architecture description languages*** (ADLs) are advocated to specify processor architectures. Often, both the processor RTL and a compiler/simulator tool chain can be generated automatically. Recent efforts attempt to automatically extract custom instructions from application, given a tight budget of available instruction slots. Because the main mechanism of application customization is through instruction extension, the amount of acceleration an ASIP can achieve is limited. Therefore, ASIP designs cannot completely replace what can be offered by high-level synthesis. Furthermore, they have to compete against the existing design ecosystem (including legacy software) made available by dominant embedded processor vendors who continually enhance their instruction sets for each processor generation.

For high-level synthesis to be successful as a component technology within an ESL design method, many advanced issues beyond the scope of this chapter have to be solved. The key drivers for these issues are that ***design productivity*** and ***quality of result*** (QoR) have to be substantially better than RTL to justify the departure from a mature, well-tested design flow.

On the design productivity front, classical high-level synthesis only tackles behavioral description with loop kernel level complexity and accepts only a subset of software program constructs. As a result, the design flow requires significant manual partitioning and rewriting effort from designers, diminishing the productivity gain promised by high-level synthesis. To succeed, high-level synthesis has to scale analysis and optimization algorithms to handle applications in their entirety and target architectures beyond FSMD. For example, performance-constrained–based algorithms work only on a basic block, and it is impractical to ask users to specify the cycle count constraint of every basic block in a complex program. Therefore, either resource-constrained algorithms driven by a design exploration environment have to be applied or new performance-budgeting algorithms capable of distributing end-to-end performance constraints to individual blocks have to be developed. As another example, classical high-level synthesis does not permit the use of pointers in the behavioral description, yet pointers are pervasively used in C/C++ programs. Pointer analysis [Hind 2000] was demonstrated to relax such limitations [Panda 2001a; Semeria 2001; Zhu 2002]. As another example, ***multi-processor system-on-chip*** (MPSOC) architectures [Dutt 2001; Helmig 2002; Intel 2002; Artieri 2003] were explored to enable coarse-grained parallelism, and both bus-based communication schemes [Pasricha 2008] and ***network-on-chips*** (NOC) [Dally 2001] were proposed to provide on-chip communication support among the processing elements.

Although it is debatable whether the new acronyms above truly advance the state-of-the-art in high-level synthesis, it is the QoR a synthesis tool can achieve that would finally earn acceptance by designers. As a discipline, high-level synthesis sits at the intersection of multiple domains, including parallelizing compilers, computer architecture, and circuit optimization. Therefore, it has to exploit and adapt existing techniques in these domains, and innovations likely result by crossing boundaries of these domains. For example, *presynthesis transformations* were shown to have significant impact on QoR [Bhasker 1990; Nicolau 1991; Gupta 2003], yet a different strategy needs to be taken from those in optimizing compilers. As another example, like in general purpose computing, memory accesses are often the performance bottleneck. The freedom in creating a customized memory system in high-level synthesis led to many innovative *memory optimization algorithms* exploiting memory access patterns at the application side and available bandwidth at the architecture and circuit level [Panda 2001b; Wolf 2003]. Another problem that led to the poor performance of classical high-level synthesis is the lack of a link to downstream logic synthesis and place-and-route tools. The classical methods abstract away the effects of downstream tools by use of area and timing

estimation models that are often too crude to be useful. This leads to the well-known timing closure problem. A promising direction is the so-called *C-to-gate methodology* in which behavioral and logic synthesis are integrated in an effective fashion. Finally, there is an urgent need to tightly couple physical design with high-level synthesis to allow for better predictability of design results at the later stages of chip design [Xu 1998; Um 2003].

5.8 EXERCISES

5.1. (Frontend/IR Design) Use lex/yacc to build a frontend for TinyC, which is given in Section 5.2.1.

5.2. (Resource-Constrained Scheduler) Implement the list scheduler with the frontend built in Problem 5.1. The resource constraint is passed as command line in the following format:

- behsyn -R constraint_spec foo.c
- constraint_spec: = component [; component]*
- component: = opcode [, opcode]* : num

For example, *behsyn -R "OP_ADD: 2; OP_MUL: 1" foo.c* specifies that two adder components (which implements OP_ADD) and one multiplier component (which implements OP_MUL) are allocated. Your program should optimize the expected cycle count under the specified resource constraint.

5.3. (Register Binder) Implement a register binder with one of the following algorithms:

- Left edge algorithm
- Coloring algorithm
- Weighted clique partitioning algorithm

Your program should optimize the number of registers used while respecting the result of scheduling in Exercise 5.2.

5.4. (Functional Unit Binder) The goal of this exercise is to implement a functional unit binder. The result of binding should respect the result of scheduling (Problem 5.2) and register allocation (Problem 5.3) in the previous exercises, while minimizing the cost of multiplexers.

5.5. (HDL Generation) The goal of this exercise is to export the result of behavioral synthesis to a form that is acceptable to commercial logic synthesis and backend tools:

- A component generator that can output VHDL/Verilog code that uses the Synopsys DesignWare components to implement the necessary RTL component in your synthesis result.
- A controller/datapath generator that outputs the VHDL/Verilog code for the controller, datapath, and the top-level design, respectively.

5.6. (Multicycling and Functional Unit Pipelining) Modify Algorithm 5.1 to incorporate realistic timing:
- Functional unit latency is larger than one.
- Functional unit latency is larger than one, but can process data every cycle.

5.7. (Register Binding) It has been shown in [Golumbic 1980] that the coloring problem can be optimally solved in linear time, if the interference graph is chordal.
- Show that the interference graph for values in a basic block is chordal.
- Show that the interference graph for values in a TinyC program is chordal.

5.8. (Phase Ordering) Create an example to demonstrate that scheduling can significantly impact the register allocation result. Devise a strategy to mitigate this so-called phase ordering problem.

ACKNOWLEDGMENTS

We thank Rami Beidas and Wai Sum Mong of University of Toronto for their help in preparing the examples used in the text. We also thank Professor Jie-Hong (Roland) Jiang of National Taiwan University, Professor Preeti Ranjan Panda of Indian Institute of Technology, Delhi, and Dr. Sumit Gupta of Nvidia, for their valuable feedback on this chapter.

REFERENCES

R5.0 Books

[Aho 2006] A. Aho, R. Sethi, J. Ullman, and M. Lam, *Compilers: Principles and Techniques and Tools,* Second, Addison-Wesley, Reading, MA, 2006.

[De Micheli 1994] G. De Micheli, *Synthesis and Optimization of Digital Circuits,* McGraw-Hill, Hightstown, NJ, 1994.

[Gajski 1992] D. D. Gajski, N. D. Dutt, A. C.-H. Wu, and S. Y.-L. Lin, *High-level Synthesis: Introduction to Chip and System Design,* Kluwer Academic, Norwell, MA, 1992.

[Gajski 1994] D. D. Gajski, F. Vahid, Narayan, and J. Gong, *Specification and Design of Embedded Systems,* Prentice-Hall, Englewood Cliffs, NJ, 1994.

[Gajski 2000] D. D. Gajski, J. Zhu, R. Domer, A. Gerstlauer, and S. Zhao, *SpecC: Specification Language and Methodology,* Kluwer Academic, Norwell, MA, 2000.

[Golumbic 1980] M. C. Golumbic, *Algorithmic Graph Theory and Perfect Graphs,* Academic Press, 1980.

[Grötker 2002] T. Grötker, S. Liao, G. Martin, and S. Swan, *System Design with SystemC,* Kluwer Academic, Norwell, MA, 2004.

[Mishra 2008] P. Mishra and N. Dutt, *Processor Description Languages,* Morgan Kauffman, San Francisco, 2008.

[Pasricha 2008] S. Pasricha and N. Dutt, *On-Chip Communication Architectures: System on Chip Interconnect,* Morgan Kauffman, San Francisco, 2008.

[Yen 1996] T.-Y. Yen and W. Wolf, *Hardware-Software Co-synthesis of Distributed Embedded Systems,* Kluwer Academic, Norwell, MA, 1996.

R5.1 Introduction

[Berry 1992] G. Berry and G. Gonthier, The Esterel Synchronous Programming Language: Design, Semantics, Implementation, *Science of Computer Programming*, 19(2), pp. 87–152, November 1992.

[Halbwachs 1991] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, The synchronous data flow programming language LUSTRE, *Proceedings of the IEEE*, 79(9), pp. 1305–1320, September 1991.

[Kahn 1974] G. Kahn, The semantics of a simple language for parallel programming, in *Information Processing*, pp. 471–475, August 1974.

[Krolikoski 1999] S. J. Krolikoski, F Schirrmeister, B. Salefski, J. Rowson, and G. Martin, Methodology and technology for virtual component driven hardware/software co-design on the system-level, in *Proc. IEEE Int. Symp. on Circuits and Systems*, 6, pp. 456–459, July 1999.

[Lee 1996] E. Lee and A. Sangiovanni-Vincentelli, Comparing models of computation, in *Proc. IEEE/ACM Int. Conf. on Computer-Aided Design*, pp. 234–241, November 1996.

[Lee 1987] E. A. Lee and D. G. Messerschmitt, Static scheduling of synchronous data flow programs for digital signal processing, *IEEE Trans. on Computers*, 36(1), pp. 24–35, January 1987.

[SIA 2007] Semiconductor Industry Association, *The International Technology Roadmap for Semiconductors*: 2007 Edition, http://public.itrs.net, 2007.

[Zhu 1997] J. Zhu, R. Doemer, and D. Gajski, Syntax and semantics of SpecC+ language, in *Proc. Seventh Workshop on Synthesis and System Integration of Mixed Technologies*, pp. 75–82, December 1997.

R5.7 Concluding Remarks

[Artieri 2003] A. Artieri, V. D'Alto, R. Chesson, M. Hopkins, and M. C. Rossi, Nomadik™ open multimedia platform for next-generation mobile devices, *STMicroelectronics Technical Article TA305*, http://www.si.com, 2003.

[Bhasker 1990] J. Bhasker and H.-C. Lee, An optimizer for hardware synthesis, *IEEE Design & Test of Computers*, 7(5), pp. 20–36, September–October 1990.

[Brewer 1988] F. D. Brewer, Constraint driven behavioral synthesis, Ph.D. thesis, Dept. of Computer Science, University of Illinois, May 1988.

[Camposano 1991] R. Camposano, Path-based scheduling for synthesis, *IEEE Trans. on Computer-Aided Design*, 10(1), pp. 85–93, January 1991.

[Dally 2001] W. J. Dally and B. Towles, Route packets, not wires: On chip interconnection networks, in *Proc. ACM/IEEE Design Automation Conf.*, pp. 684–689, June 2001.

[De Man 1986] H. De Man, J. Rabaey, P. Six, and L. Claesen, Cathedral-II: A silicon compiler for digital signal processing, *IEEE Design & Test of Computers*, 3(6), pp. 73–85, November-December 1986.

[Dutt 2001] S. Dutt, R. Jensen, and A. Rieckmann, Viper: A multiprocessor SOC for advanced set-top box and digital TV systems, *IEEE Design & Test of Computers*, 18(5), pp. 21–31, September–October 2001.

[Ernst 1996] R. Ernst, J. Henkel, T. Benner, W. Ye, U. Holtmann, D. Herrmann, and M. Trawny, The COSYMA environment for hardware/software cosynthesis of small embedded systems, *J. Microprocessors and Microsystems*, 20(3), pp. 159–166, May 1996.

[Fauth 1995] A. Fauth, J. Van Praet, and M. Freericks, Describing instruction set processors with nML, in *Proc. IEEE/ACM Design, Automation and Test in Europe Conf.*, pp. 503–507, March 1995.

[Gomez 2004] J. I. Gomez, P. Marchal, S. Verdoorlaege, L. Pinuel, and F. Catthoor, Optimizing the memory bandwidth with loop morphing, in *Proc. IEEE Int. Conf. on Application-Specific Systems, Architectures and Processors*, pp. 213–223, September 2004.

[Grun 2001] P. Grun, N. Dutt, and A. Nicolau, APEX: Access pattern based memory architecture exploration, in *Proc. Int. Symp. on System Synthesis*, pp. 25–32, September 2001.

[Gupta 1992] R. K. Gupta, C. N. Coelho, and G. De Micheli, Synthesis and simulation of digital systems containing interacting hardware and software components, in *Proc. ACM/IEEE Design Automation Conf.*, pp. 225–230, June 1992.

[Gupta 2003] S. Gupta, N. D. Dutt, R. K. Gupta, and A. Nicolau, SPARK: A high-level synthesis framework for applying parallelizing compiler transformations, in *Proc. IEEE Int. Conf. on VLSI Design*, pp. 461–466, January 2003.

[Gyrczyc 1984] E. Gyrczyc, Automatic generation of micro-sequenced data paths to realize ADA circuit descriptions, Ph.D. thesis, Carleton University, 1984.

[Halambi 1999] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. D. Dutt, and A. Nicolau, EXPRESSION: A language for architectural exploration through compiler/simulator retargetability, in *Proc. IEEE/ACM Design, Automation and Test in Europe Conf.*, pp. 485–490, March 1999.

[Helmig 2002] J. Helmig, Developing core software technologies for TI's OMAP™ platform, *Texas Instruments*, http://www.ti.com, 2002.

[Hind 2000] M. Hind and A. Pioli, Which pointer analysis should I use?, in *Proc. ACM SIGSOFT Int. Symp. on Software Testing and Analysis*, pp. 113–123, August 2000.

[Hoffmann 2001] A. Hoffmann, O. Schliebusch, A. Nohl, G. Braun, O. Wahlen, and H. Meyr, A methodology for the design of application specific instruction set processors (ASIP) with the machine description language LISA, in *Proc. IEEE/ACM Int. Conf. on Computer-Aided Design*, pp. 625–630, November 2001.

[Intel 2002] Intel, *Product Brief: Intel IXP2850 Network Processor*, http://www.intel.com, 2002.

[Kurdahi 1987] F. J. Kurdahi and A. C. Parker, REAL: A program for register allocation, in *Proc. ACM/IEEE Design Automation Conf.*, pp. 210–215, June 1987.

[Landskov 1980] D. Landskov, S. Davidson, B. Shriver, and P. W. Mallett, Local microcode compaction techniques, *ACM Computing Surveys*, 12(3), pp. 261–294, September 1980.

[Marwedel 1986] P. Marwedel, A new synthesis for the MIMOLA software system, in *Proc. ACM/IEEE Design Automation Conf.*, pp. 271–277, June 1986.

[McFarland 1978] M. C. McFarland, The Value Trace: A database for automated digital design, Technical Report DRC-01-4-80, Design Centre, Carnegie-Mellon University, December 1978.

[Nicolau 1991] A. Nicolau and R. Potasman, Incremental tree height reduction for high-level synthesis, in *Proc. ACM/IEEE Design Automation Conf.*, pp. 770–774, June 1991.

[Panda 2001a] P. R. Panda, L. Semeria, and G. De Micheli, Cache-efficient memory layout of aggregate data structures, in *Proc. Int. Symp. on System Synthesis*, pp. 101–106, September 2001.

[Panda 2001b] P. R. Panda, F. Catthoor, N. D. Dutt, K. Danckaert, E. Brockmeyer, C. Kulkarani, A. Vandercappelle, and P. G. Kjeldsberg, Data and memory optimization techniques for embedded systems, *ACM Trans. on Design Automation of Electronic Systems*, 6(2), pp. 149–206, February 2001.

[Pangrle 1987] B. M. Pangrle and D. D. Gajski, Slicer: A state synthesizer for intelligent silicon compilation, in *Proc. IEEE/ACM Int. Conf. on Computer-Aided Design*, pp. 42–45, November 1987.

[Parker 1986] A. C. Parker, J. Pizarro, and M. Mlinar, MAHA: a program for datapath synthesis, in *Proc. ACM/IEEE Design Automation Conf.*, pp. 461–466, June 1986.

[Paulin 1989] P. Paulin and J. Knight, Force-directed scheduling for the behavioral synthesis of ASIC's, *IEEE Trans. on Computer-Aided Design*, 8(6), pp. 661–679, June 1989.

[Semeria 2001] L. Semeria and G. De Micheli, Resolution, optimization, and encoding of pointer variables for the behavioral synthesis from C, *IEEE Trans. on Computer-Aided Design*, 20(2), pp. 213–233, February 2001.

[Thomas 1988] D. E. Thomas, E. M. Dirkes, R. A. Walker, J. V. Rajan, J. A. Nestor, and R. L. Blackburn, The system architect's workbench, in *Proc. ACM/IEEE Design Automation Conf.*, pp. 337–343, June 1988.

[Tseng 1986] C.-J. Tseng and D. P. Siewiorek, Automated synthesis of data paths in digital systems, *IEEE Trans. on Computer-Aided Design*, 5(3), pp. 379–395, March 1986.

[Um 2003] J. Um and T. Kim, Synthesis of arithmetic circuits considering layout effects, *IEEE Trans. on Computer-Aided Design*, 22(11), pp. 1487–1503, November 2003.

[Wolf 2003] W. Wolf and M. Kandemir, Memory system optimization of embedded software, *Proceedings of The IEEE*, 91(1), pp. 165–182, January 2003.

[Wuytack 1999] S. Wuytack, F. Catthoor, G. D. Jong, and H. J. De Man, Minimizing the required memory bandwidth in VLSI system realizations, *IEEE Trans. on Very Large Scale Integration Systems*, 7(4), pp. 433–441, April 1999.

[Xu 1998] M. Xu and F. J. Kurdahi, Layout-driven high level synthesis for FPGA based architectures, in *Proc. IEEE/ACM Design, Automation and Test in Europe*, pp. 446–450, February 1998.

[Zhu 2002] J. Zhu, Symbolic pointer analysis, in *Proc. IEEE/ACM Int. Conf. on Computer-Aided Design*, pp. 150–157, November 2002.